Andreas Spillner · Tilo Linz · Hans Schaefer

Software Testing Foundations

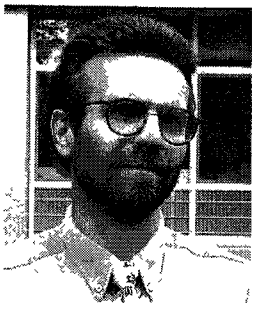
A Study Guide for the Certified Tester Exam

- Foundation Level
- ISTQB compliant

dpunkt.verlag

ISTQB

About the Authors



Andreas Spillner is professor for Computer Science at the Department of Electrical Engineering and Computer Science of 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 member of the German Testing Board. His work emphasis is in software, quality assurance, testing and object-oriented system development.

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, from 2002 to 2005, president of the ISTQB. His work emphasis is consulting and coaching projects about software quality management and optimizing software development and testing processes.

Hans Schaefer is 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 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

dpunkt.verlag

Andreas Spillner
spillner@informatik.hs-bremen.de

Tilo Linz
tilo.linz@imbus.de

Hans Schaefer
hans.schaefer@ieee.org

Editor: Christa Preisendanz
Copyeditor: Angelika Shafir, Tel Aviv, Israel
Producer: Josef Hegele, Dossenheim
Cover Design: Helmut Kraus, www.exclam.de
Printer: Koninklijke Wohrmann B.V., Zutphen, Netherland

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data are available in the Internet at <http://dnb.ddb.de>

ISBN 3-89864-363-8

1st Edition

Ringstraße 19 b
69115 Heidelberg

This 1st English book edition conforms to the 3rd German edition
Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard (dpunkt.verlag GmbH, ISBN 3-89864-358-1), which was published in August 2005

5 4 3 2 1 0

Foreword by Rex Black

I've been in the software and systems engineering business for over twenty years, with most of that time spent as a tester. In the 1980s, when I shifted my career emphasis from programming and system administration to testing, the resources were pretty limited. There were a few books by testing pioneers such as Boris Beizer, Bill Hetzel, Glenford Myers, and Bill Perry. Some of these books were – and remain – good resources. Someone new to the field could cover the entire software and systems testing library in a few months of reading.

Not any more. Now we have dozens and dozens of books out there on testing. You can read a book on testing specific kinds of applications, like Web applications and embedded applications. You can read a book on testing in formal settings and informal settings. You can even read a book or two on test management.

However, every professional needs to start somewhere. Every profession needs its foundation. The profession of software and systems testing needs books that provide the basic techniques, tools, and concepts. This is one such book.

This book will provide you with a solid practical foundation for your work and study of testing. Software and system testing suffers from a serious gap between best practices and common practices. If you're someone who is making a living from doing testing but haven't gotten around to reading a book, why not start with this one?

The authors wrote this book using the International Software Test Qualification Board's Foundation Level Syllabus as an outline. So, if you're pursuing test certification, I recommend this book. You can get certified according to the Foundation Level Syllabus by taking an exam offered through a recognized National Board. Such National Boards include the American Testing Board, the Indian Test Board, and the Israeli Test Certification Board, to name three such boards that I serve on.

This book would also make a fine textbook for a college course. If you're a professor looking for a good testing textbook, both you and your students may find this book a good choice.

This book should prove especially useful to you if you work on in-house system development. The discussion on the role, techniques, and importance of requirements specification and acceptance testing in such environments is excellent. Of course, we don't always find ourselves working in organizations that have the overall system lifecycle process maturity that underpins this book. However, assuming that the testing

process is part of a larger set of mature, well-managed development, maintenance, and deployment processes is a smart way to keep the book from spiraling into a complex discussion on how testing can adapt to dysfunctional organizations.

One problem we face in the testing profession is the lack of a universally-accepted glossary. That leads to a lot of discussion and confusion. The authors deal with that by providing definitions for their terms, based on the International Software Testing Qualification Board's glossary.

I found a lot that I liked in this book. It provides a good description of what software and systems testing is. It explains not just the best practices and techniques, also the whys and hows of these techniques.

If you've read my books, *Critical Testing Processes* and *Managing the Testing Process*, you know that I like case studies and examples. If you've taken my training courses, you've worked through exercises based on real-world examples. This book uses a well-described, practical, true-to-life running case study to illustrate the key points. That helps bring the material to life and make it clear.

I also liked the survey of the commonly-used and commonly-useful black box and white box techniques. The authors also provide good brief discussions of some of the more unusual – but sometimes useful – techniques, too. If you're an analyst or test manager, this should help you understand the essential techniques of test design.

There's also a good survey of test automation tools and techniques. The authors give a balanced perspective that neither bashes nor boosts the tools. With so much hype and confusion surrounding this topic – and, sadly, so many failed attempts at test automation – the authors' dispassionate approach, with plenty of cautionary notes, is refreshing.

Finally, it's nice to see a test book that includes a broad, helpful discussion of test management. Other than my own two books, this topic hasn't gotten much attention. If you're a tester or QA analyst who needs to understand the management perspective, this book should help.

As you can see, this book introduces many topics in the field of software and system testing. In spite of that, this is a relatively short book, which makes it more approachable for busy test professionals. As a writer, I know it's hard to write books that are both comprehensive and brief. The authors have struck a good balance in the level of detail they provide, focusing on the needs of the target audience. This book will provide a solid foundation for you when you read more advanced books on specific topics like test management, test design, test automation, or testing particular kinds of applications.

Bulverde, Texas, November 2005
Rex Black, President of the ISTQB

Contents

Foreword

In the foreword to the German 1st edition we had asked if any more book on software testing was needed. As both the 1st and 2nd German edition are sold out, this question can considered to have been answered positively by our readers.

The current syllabus

For the ISTQB® Certified Tester, Foundation Level, there is only one single internationally recognized syllabus since 2005. The two existing compatible syllabi by the Information Systems Examinations Board (ISEB, [URL: ISEB]) and the German Testing Board (GTB) have been combined and updated.

This 1st English book edition conforms to the 3rd German edition and conforms to the International Software Testing Qualifications Board (ISTQB, [URL: ISTQB]) syllabus published in July 2005. The book includes the additions made in the syllabus with respect to the two earlier versions.

What is new in the new ISTQB syllabus?

Besides the addition of new contents (amongst others the test first approach and risk-based testing), the formulation of learning objectives is basis for a new learning dimension. Explicit learning objectives on the one hand make it easier for the reader to keep oriented. They also make it more transparent which knowledge to which depth is expected from an ISTQB Certified Tester – Foundation Level. This means, for example, that at the lowest level of the learning objectives the glossary definitions of all marked key terms of the syllabus are relevant for the examination. "The terms are not defined any more in the new ISTQB Foundation Level Syllabus, but in the ISTQB Glossary of Testing Terms [URL: ISTQB] and its national equivalents. The new ISTQB Foundation Level Syllabus is a self-containing whole explaining Best-Practices of Software Testing in the syllabus itself in a consistent level of detail. The explanation is not located in outside sources like national standards. Analogous to the Advanced Level the structuring of the contents relevant for the examination (for example test management) has now also been applied in the ISTQB Foundation Level. Thus the Foundation Level already creates the basis for the additional knowledge for the Advanced Level" (Horst Pohlmann, German Testing Board, Working Party Foundation Level).

Certification

The education and certification for the Certified Tester have been very well received worldwide. At the beginning of 2005, there were

already more than 20 000 certified testers, i.e. people who have passed the exams organized worldwide by several national testing boards [URL: ISTQB]. Ca. 80 % of the examined people passed the exam and got the certificate. The official exam questions are currently being updated to match the new ISTQB syllabus. After a time period for adaptation, new examinations are now run only based on the new syllabus. Several training companies are accredited for running seminars to prepare people for the Certified Tester examination. Thus qualified trainings are available, for example in Europe, USA or India.

ISTQB members

Just now the following countries have national testing boards in the ISTQB: Austria, Denmark, England, Finland, France, Germany, India, Israel, Japan, Korea, Netherlands/Belgium, Norway, Poland, Portugal, Spain, Sweden, Switzerland as well as the USA and Australia/New Zealand.

In order to serve the international interest about the topic of software testing and the Certified Tester education, we published a Dutch edition in 2004. Translations into Polish and Romanian are under way.

The next qualification level

At the 2nd level, there are currently two schemes and two syllabi: ASQF/ iSQI Advanced Level (developed by German Testing Board) and ISEB Practitioner Level (developed by UK Board). Both are recognized by ISTQB as professional qualifications for testers as they have gained a respect over many years in the testing community. ISTQB intends that these two 2nd level schemes will be integrated into a future single unified 2nd level "ISTQB Advanced Level" qualification which should supersede both existing schemes.

Seminars for advanced topics (for example test management, test methods) are already being offered and well frequented. We are currently writing the literature matching this syllabus. The book "Praxiswissen Testmanagement" (in German) will be published in spring 2006, books about "Test methods" and the topics for the yet to be defined Expert Level shall follow.

Use at universities and colleges

We are pleased to note that the contents of this book have been adopted at universities and colleges. Lectures with attached examination have been and are successfully offered at the technical universities of Munich and Darmstadt, the university of Dortmund, the universities of applied science in Cologne and Bremen as well as at the university of Iceland in Reykjavik. The students were able to conduct the exam to the Certified Tester, Foundation Level. The authors are ready to supply interested university teachers with additional supporting material.

Thank you notes

We want to thank all readers for helpful comments, which have contributed to correcting faults and amending unclear items in the 1st and 2nd German edition of this book. We want to especially thank the

colleagues at Siemens Austria. The discussion with them in order to design the test cases for boundary value analysis has led to a total redesign of tables 5–8 and 5–9 in chapter 5.

A further thank you to the colleagues in the GTB and ISTQB, without whose great work there would be no Certified Tester scheme. We want to especially thank Horst Pohlmann for his excellent contributions when composing the syllabi, examination questions and the Certified Tester glossary.

We want to cordially thank Martin Pol for his translation of the book to Dutch. Rex Black has also given us many valuable comments and written a foreword for this English edition.

Andreas Spillner, Tilo Linz and Hans Schaefer
Bremen, Möhrendorf and Valestrandsfossen,
January 2006

1 Introduction

Software has found an enormous dissemination in the past years. There are almost no machines or facilities left that are not controlled by software or at least include software. In automobiles, for example, from the engine management through transmission control up to the brake assistant, more and more control functions are taken over by microprocessors and are controlled by software. Thus, software cru cially contributes to the functionality of devices and plants. Likewise, the smooth operation of an enterprise or an organization depends largely on the reliability of software systems that are used for supporting the business processes or particular tasks. How fast an insurance company is able to introduce a new product or at least a new rate today depends most likely on how fast the IT-systems can be adjusted or extended.

High dependence on the correct functioning of the software

Within both sectors (embedded and commercial software systems), the quality of software became the most important factor for the success of products or enterprises.

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 systematic evaluation and testing of the developed software. Appropriate procedures have, in some cases, found their way into the daily practice of software development. But in many sectors, there is still a big need of knowledge transfer about evaluation and testing procedures.

Basic knowledge for structured evaluation and testing

With this book, we offer basic knowledge that helps to achieve structured and systematic evaluation and testing. This should contribute to an improved quality of the software. The content of this book is written in a way that does not subsume previous knowledge of software quality assurance. The book is designed as a textbook and is meant for self-studies. A continuous case example is included in order to help understand every shown topic and its practical solution.

We want to appeal to the software testers in software and industry enterprises, who want to achieve a well founded basic knowledge of the principles behind testing. We also address programmers and developers who are already practicing testing tasks or will do so. The book will help project managers and team leaders to improve the effectiveness and efficiency of software tests. Even people in related disciplines

near IT jobs and other employees who are involved in the process of acceptance, introduction and further development of IT applications, will find help for their daily tasks in this book.

Evaluation and testing procedures have a high cost in practice (expenditures in this sector are estimated with 25 % up to 50 % of the software development time and costs [Koomen 99]). Still, there are almost no universities, colleges or vocational schools in the sector of computer science that offer courses that intensively treat those problems. Both students and teachers should use this book. It provides the material for a basic course.

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

Certification program for software testers

In 1998, the *Information Systems Examinations Board* [URL: ISEB] of the *British Computer Society* [URL: BCS] [URL: ISEB] started such a certification scheme.

International initiative

Similar to the British example, other countries took up these activities and established country specific *Testing Boards*, in order to make it possible to run training and examination in the language of the respective country. These national boards cooperate in the *International Software Testing Qualifications Board* [URL: ISTQB].

The current structure of the ISTQB is shown in figure 1-1.

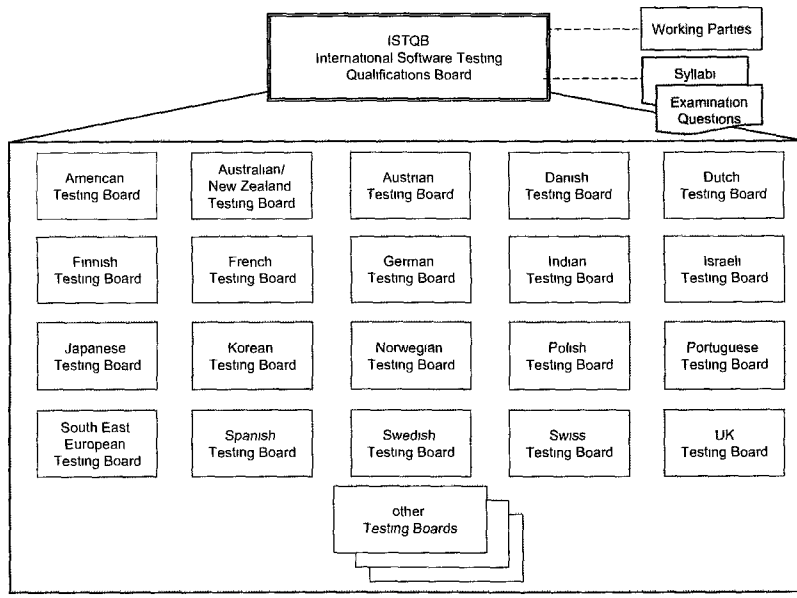


Fig. 1-1 *International Software Testing Qualification Board*

The *International Software Testing Qualifications Board* coordinates the national initiatives and provides the uniformity and comparability of the teaching and exam contents in the countries involved.

National Testing Boards

The *National Testing Boards* are responsible for issuing and maintaining curricula in their country language 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 of the seminars. The seminar participants get, after passing an exam, an internationally recognized qualification certificate.

The *ISTQB Certified Tester* Qualification scheme has three steps [URL: ISTQB]. The basics are described in the curriculum (syllabus) for the *Foundation Level*. Building on this is the *Advanced Level* certificate, showing a deeper knowledge of testing. The third level, the *Expert Level* certificate, is in process.

The content of this book corresponds to the requirements of the ISTQB *Foundation Certificate*. The knowledge needed for the exams can be acquired by self-studies. The book can also be used to extend the knowledge after or parallel to participation in a course.

Chapter overview basics

The topics of this book and thus the rough structure of the course contents for the *Foundation Certificate* are described below.

In chapter 2, we will discuss the basics of software testing. Besides the motivation, when to test with which goals and how intensively, the concept of a basic test process will be shown. It will deal with the psychological difficulties experienced when testing one's own software, and the blindness for one's own →errors.

Testing in the software life cycle

Chapter 3 discusses which test activities should be done during the software developing process, and how they relate to other development tasks. In addition to the different →test levels and -phases it will deal with the differences between functional and nonfunctional test. The economy of testing and how to test changes as well as testing in maintenance will also be discussed.

Static test

Chapter 4 discusses static methods, i.e. procedures where the test object is analyzed but not executed. Reviews and static analyses are already used by many enterprises with good experiences. The different methods and techniques will be described.

Dynamic test

Chapter 5 deals with testing in a narrower sense. The classification of dynamic testing into *black box* and *white box* techniques will be discussed. Different test techniques are explained in detail with the help of a continuous example. At the end of the chapter, we will illustrate the reasonable usage of exploratory and intuitive test, to be used in addition to the other techniques.

Test management

Chapter 6 shows which aspects should be considered in test management, how systematic incident handling looks like and some basics about establishing sufficient →configuration management.

Test tools

Testing of software without an appropriate support of tools is very labor and time intensive. The seventh and last chapter of this book introduces different classes of tools for supporting testing and hints for tool selection and implementation.

Notes to the subject matter and for the exam are in the appendix

The appendix gives notes and extra information to the subject matter and for the exam to the Certified Tester. Further appendices of this book contain explanations to the test plan according to [IEEE 829-1983], exemplary exercises, a glossary and the list of literature. The technical terms are marked with an appropriate hint when they appear the first time in the text. The hint points to a detailed definition in the glossary. Text passages that go beyond the material of the syllabus are marked as *excursions*.

2 The Basics of Software Testing

This chapter will explain some basic facts of software testing, everything that is required for understanding the following chapters. Important phrases and essential vocabulary will be explained using an example application. This example appears frequently to illustrate and motivate the subject matter throughout the book. In the following section, this example will be introduced. The fundamental test process and the single activities of testing will be illustrated. Psychological problems will de discussed.

The procedures for testing software presented in this book are mainly illustrated by one general example. The fundamental scenario is described as follows:

Case study "VirtualShowRoom" - VSR

A car manufacturer develops a new electronic sales support system called *VirtualShowRoom* (VSR). The final version of this software system is supposed to be installed at every car dealer worldwide. Any customer, who is interested in buying a new car, will be able to configure her 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 accurate price of the configured car. This functionality will be implemented by a sub-system called *DreamCar.*

If the customer has made up her mind, she will be able to calculate the most suitable payment (*EasyFinance) as* well as to place the order online (*JustInTime*). Of course, she will get the possibility to sign up for the appropriate insurance (*NoRisk*). Personal information and contract data about the customer is managed by the *ContractBase* sub-system.

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

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

Before shipping the VSR-System, it must be tested thoroughly. The project members, who have been assigned for testing the software, apply different techniques and procedures. This book contains the basic knowledge for those techniques and procedures of software testing.

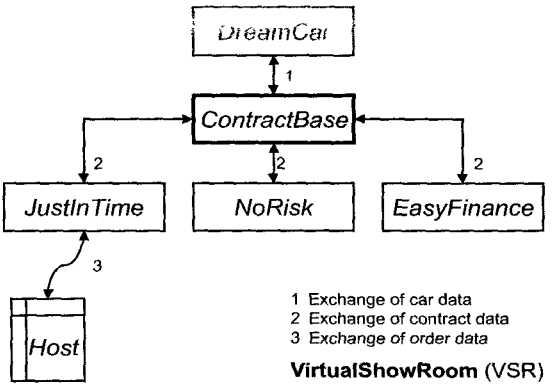


Fig. 2-1 *Architecture of the VSR-System*

2.1 Terms and Motivation

Requirements

During the construction of an industry product, the parts and the final product are usually tested to check if they fulfill the given →requirements. It will be examined if the product solves the required task. There may be differences between the requirements and the implemented product. If the product exhibits problems, necessary corrections must be made in the production process and/or in the construction.

Software is immaterial

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

The dynamic behavior of the software can however not be checked this way. It must be done through →testing, where the tested software will be executed on a computer. Its behavior must be compared to the given requirements. Thus, the testing of software is a very important and difficult task in software development process. It contributes to reducing the →risk of use of the software, because bugs can be found by testing. Testing and its documentation are sometimes also required by the contract or in legal or industrial standards.

Example

To identify and repair possible faults before delivery, the VSR-System from the case example must be tested intensively before it is used. For example, if the system executes order transactions wrong, this could result in frustration for the customer and a serious financial or image loss to the dealer and the

car manufacturer. Anyway, not finding the bugs holds a high risk when using the system.

2.1.1 Error and Bug Terminology

What is an error, failure or fault?

When do we have a behavior of the system, which does not conform to requirements? A situation can be classified as incorrect only after we know how the expected correct situation is supposed to look like. Thus, a →failure is a non-fulfillment of a given requirement, a discrepancy between the →actual result or behavior (identified while executing the test) and the →expected result or behavior (defined in the specifications or requirements). A failure is present if a warrantable (user) expectation is not fulfilled adequately. Examples for failures are products that are too hard to use, or too slow, but still fulfill the functional requirements.

Causal chain

In contrast to physical systems software failures do not occur because of aging or abrasion. They occur because of →faults in the software. Every fault (or →defect or →bug) in the software is there since it was developed or changed. Yet, the fault materializes only when executing the software, becoming visible as a failure.

Failures

To describe this circumstance that a user experiences a problem, [IEEE 610.12] uses the term failure. However, other terms like "problem" or "incident" are often used. During testing or use of the software, the failure becomes visible to the →tester or user. For example an output is wrong or the application crashes.

Faults

We must differentiate between the occurrence of a failure and its cause. A failure has its roots in a fault in the software. This fault is also called defect or internal error. Programmer slang for this term is "bug". For example wrongly programmed or forgotten code in the application.

Defect masking

It is possible that a fault is hidden by one or more other faults in different parts of the application (→defect masking). In that case, a failure only occurs after the masking defects have been corrected. This shows 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 bad example is some small corruption of stored data, which may be found long time after it first occurred.

The cause of a fault or defect is an error or mistake by a person. For example, wrong programming by the developer, or misunder-

standing the commands in a programming language. However, faults may even be caused by environmental conditions, like radiation, magnetism etc., introducing hardware problems. This last factor is, however, not further discussed in this book.

More detailed descriptions of the terms within the domain of testing are given in the following paragraphs.

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. First, we only know the effect of a defect but not the precise location in the software. The localization and the correction of defects are the job of the software developer and are called →debugging. Repairing a defect generally increases the →quality of the product, provided that in most cases no new defects are introduced. Debugging is often equated with testing. But testing and debugging are totally different activities.

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

A test is a sample examination

Every execution (even using more or less random samples) of a →test object in order to examine it is testing. The →conditions for the test must be defined. The actual and the expected behavior of the test object must be compared[1].

Testing software has different purposes:

Executing a program in order to find failures
Executing a program in order to measure quality
Executing a program in order to provide confidence[2]
Analyzing a program or its documentation in order to prevent defects

The whole process of systematically executing programs to demonstrate the correct implementation of the requirements, to increase confidence and detect failures is called test. In addition, test includes static methods, i.e. static analysis of software products using tools, as well as document reviews (see chapter 4).

Besides the execution of the test object with →test data, the planning, design, implementation and analysis of the test (→test manage-

1. It is not possible to prove correct implementation of the requirements. We can only reduce the risk of serious bugs remaining through testing.
2. If a thorough test finds little or no failures, confidence in the product will increase.

ment) are also part of the →test process. A →test run or →test suite consists of the execution of one or more →test cases. A test case contains defined test conditions (mostly the requirements for execution), the inputs and the expected outputs or the expected behavior of the test object. A test case should have a high probability to reveal hitherto unknown faults [Myers 79].

Several test cases can often be combined to →test scenarios, whereby the result of one test case is used as the starting point for the next test case. For example, a test scenario for a database application can contain one test case, which writes a date into the database, another test case, which manipulates that date and a third test case, which reads the manipulated date out of the database and deletes it. Then all the three test cases will be executed, one after another, all in a row.

No complex software system is bug free

At present, there is no known bug free software system and there will not be in the near future, once the system has nontrivial complexity. Often the reason for a fault is that certain exceptional cases were not considered during development as well as during testing of the software. Be it the incorrectly calculated leap year or the not considered boundary condition for the timely response or the needed resources. On the other hand there are many software systems in many different fields that operate reliably, day in and out.

Absolute correctness can not be achieved with testing

Even if all the executed test cases do not reveal any further failures, we cannot conclude with complete safety (except for very small programs) that there do not exist further faults.

Excursion: Naming of tests

There are many confusing terms for different kinds of software testing tasks Some will be explained later within the description of the different test levels (see chapter 3) This excursion is supposed to explain some of the different terms It is helpful to differentiate these categories of testing terms

1 **→Test objective or test type:** Calling a kind of test by its purpose (e g →load test)
2 **→Test technique:** The test is named using the name of the technique used for the specification or execution of the test (e g →business-process-based test or →boundary value test)
3 **Test object:** The test is named after the kind of the test object to be tested (e g GUI test or DB test (data base test))
4 **Testing level:** The test is named after the level or phase of the underlying life cycle model (e g →system test)
5 **Test person:** The test is named after the person subgroup executing the tests (e g developer test, →user acceptance test)
6 **Test extent:** The test is named after the level of extent (e g partial →regression 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

Testing of software contributes to improvement of the →software quality. This is done through identifying defects and their following correction by debugging. But testing is also measurement of software quality. 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 entails more than just the elimination of failures that occurred during the testing. According to the ISO/IEC-Standard 9126-1 [ISO 9126] the following factors belong to software quality: →Functionality, →reliability, usability, →efficiency, →maintainability and portability.

All these factors or →quality characteristics (also: →quality attribute) have to be considered while testing, in order to judge the overall quality of a software product. It should be defined in advance which quality level the test object is supposed to show for each characteristic. The achievement of these requirements must then be examined with capable tests.

Example VirtualShowRoom

In the example of the VSR-System, the customer must define which of the quality characteristics are most important to her. Those have to be implemented and examined in the system. The characteristics 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.

Functionality

We discuss the individual quality characteristics of ISO/IEC-Standard 9126-1 [ISO 9126] shortly in the following. Functionality contains all characteristics, which describe the required capabilities of the system. The capabilities are usually described by a specific input/output behavior and/or an appropriate reaction to an input. The goal of the test is to prove that every single required capability in the system was implemented in the specified way. According to ISO/IEC-Standard 9126-1, the functionality characteristic contains the subcharacteristics: adequacy, interoperability, correctness and security.

An appropriate solution is given if all required capabilities exist in the system and they work adequately. Thereby it is clearly important to pay attention to, and thus to examine during testing, that the system generates the right or specified outputs or effects.

Software systems must interoperate with other systems, or 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 the previously existing system. Trouble with this cooperation should be detected by the test.

Security

Part of functionality is the fulfillment of application specific standards, agreements or legal requirements and similar regulations. Many Applications give a high importance to the aspects of access- and data security. It must be proven that an unauthorized access to applications and data, both accidentally and intentionally, will be prevented.

Reliability

Reliability describes the ability of a system to keep functioning under specific use over a specific period. In the standard, the quality 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 recover from faults in cases of software faults or of infringement of its specified interface.

Recoverability is the capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure. Following a failure, a software product will sometimes be "down" for a certain period of time, the length of which is assessed by its recoverability. The ease of recovery and the required work should also be assessed.

Usability

Usability is very important for interactive software systems. Users will not accept a system that is hard to use. How big is the expense that is required for the usage of the software, for the different user groups? Understandability, ease of learning, operability and attractiveness, as well as compliance to standards, conventions, style guides or user interface regulations are partial aspects of usability. Those quality characteristics are examined in →non-functional tests (see chapter 3).

Efficiency

The test for efficiency measures the required time and consumption of resources for the fulfillment of tasks. Resources may include other software products, the software and hardware configuration of the system, and materials (e.g. print paper, network, and storage).

Changeability and portability

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

Sub characteristics of maintainability are analyzability, changeability, and stability against side effects, testability and compliance to standards.

Maintainability

Adaptability, ease of installation, conformity and interchangeability have to be considered for the portability of software systems. Many of the aspects of maintainability and portability can only be examined by →static analysis (see chapter 4.2).

Portability

A software system cannot fulfill every quality characteristic equally well Sometimes it is possible that a fulfillment of 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 used platform to improve the efficiency, which in turn affects the portability in a negative way.

Prioritize quality characteristics

The quality characteristics must therefore be prioritized. This definition also acts as a guide for the test to determine the examination's intensity for the different quality characteristics The next chapter will discuss the amount of work for all this.

2.1.4 Test Effort

Complete testing is not possible

Testing cannot prove the absence of faults. In order to do this, a test would need to execute a program in every possible way with every possible data value. In practice, a →complete or exhaustive test is not feasible Such a test contains the combination of all possible inputs under consideration of all different conditions that have an influence on the system. Through the multiple combinatorial possibilities, the outcome of this is an almost indefinite number of tests. Such a "testing" for all combinations is not possible.

Example

This circumstance is illustrated by an example of →control flow based testing [Myers 79]

A small program with an easy →control flow will be tested The program consists of four links (IF-instructions) that are partially nested The appropriate →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 fully tested in relation to the different control flow based possibilities, every possible combination of links must be executed

At a loop limit of maximum 20 cycles, under consideration that all links are independent, the outcome is the following calculation:

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

Whereby 5 is the number of possible ways within the loop. 5^1 test cases result through the execution of every single possible way within the loop, but in each case without return to the loop starting point. If executed test cases result in one single return to the loop stating point, then $5*5 = 5^2$ different possibilities of executions must be considered etc. To that effect the total result of this calculation is about 100 quadrillions different sequences of the program.

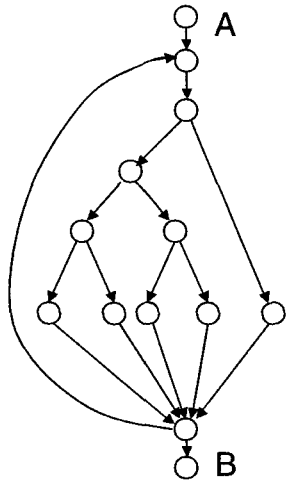


Fig. 2–2
Control flow graph of a small program

Assumed that the test is done manually and a test case, like [Myers 79] describes, takes 5 minutes to specify, to execute and to be analyzed, the time for this test would be one billion years. If we would take five microseconds instead of five minutes, because the test mainly runs automatic, 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 takes a lot 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. In the following, some example data from projects of one large German software company are shown. They 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 ten person-years' effort, coding and testing together used 40 %, and a further 8 % was used for the integration. At test intensive projects (e.g. year 2000 bug), the testing effort increased up to 80 %.

In one project, the testing effort was 1.2 times as high as the coding effort. 2/3 of the test effort was →component testing.
At another project of the software company, the system test cost was 51.9 % of the project resources.

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

Faults can cause high costs

But is this high testing effort affordable and justifiable? The counter question from Jerry Weinberg is: "Compared to what?" [DeMarco 93]. He points with that to the risks of faulty software systems. The risk is calculated as the probability of occurrence and the expected amount of loss. Faults that were not found during testing can cause high costs when the software is used. The German Newspaper *Frankfurter Allgemeine Zeitung* from the 17th January 2002 had an article with the topic "IT system break downs cost many millions". One hour system break down in the stock exchange is estimated to cost 7.8 million US $. When →safety critical systems fail, it is possible that life and health of people are 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[3] are lower than the costs of failure in operation" [Koomen 99]. Thus, the test effort is always dependent on an estimation of the application risk.

Example of a high risk in case of a failure

In case of the VSR-System, the prospective customer shall configure her 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 cheaper price. The total loss can amount to millions, depending on how much the price was miscalculated by the VSR-System for each car. The legal view is that a valid sales contract with the listed price is initiated by the online-order.

Systems with high risks must be tested more thoroughly than systems that do not generate a big loss in case of a failure. The risk assessment must be done for the individual system parts or even for single error possibilities. In the case of a high risk for failures by a system or subsystem, there must be a greater testing effort than for less critical

3. The cost must include all aspects of it, even the possible cost of bad publicity, litigation etc. Not just the cost of correction, retesting and distribution.

(sub-)systems. International standards for production of safety critical systems use this approach to require different test techniques applied for software of different integrity levels.

For a producer of a computer game, an erroneous saving of game scores can mean a very high risk, because the customers will not accept the defective game. This leads to high losses of sales, maybe even for all games of the company.

Define test intensity and -extent in dependence to the risk

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

Select adequate test procedures

There exist many different methods and procedures for testing software. Every method considers and examines particular aspects of the test object especially intensively. 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 usage and flow of data. Every test technique has its strengths and weaknesses in finding different kinds of faults. There is no test technique, which 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 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 only provide functions that are required.

→Test case explosion

The testing effort can grow large. Test managers have the dilemma that the possible test cases and test case variants quickly add to hundreds or thousands of tests. This problem is also described with the term combinatorial explosion.

Besides the necessary restriction in the number of test cases, the test manager normally has to struggle with another problem, the lack of resources.

Limited resources

Every software development project has restricted resources. Often, there will be changes in resource estimation and use during the process. There easily starts a fight for resources. The complexity of the development task is underestimated, the development team is delayed,

the customer pushes for earlier release, and the project leader wants to deliver "something" as soon as possible. The test manager normally has the worst position in this "game": Often there is only a small time window just before delivery and some few testers for running the test. It is quite sure that the test manager then does not have the time and resources for executing "astronomically" many test cases.

However, it is expected from the test manager that she delivers trustworthy results and makes sure that the software is sufficiently tested. Only if the test manager has a well-planned efficient strategy, she has the 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. For example →reviews (see chapter 4).

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

2.2 The Fundamental Test Process

Excursion: Life cycle models

In order to accomplish a structured and controllable software development, →software development models and →development processes are used There are many different models Examples are the Waterfall-model [Boehm 73, 81], the →V-model [Boehm 79] [IEEE/IEC 12207], the German V-model (V-model – Development Standard for IT Systems of the Federal Republic of Germany ('Vorgehensmodell des Bundes und der Lander") [URL V-model XT]), the Spiral Model, different incremental or evolutionary models and the agile" or 'light weight methods like XP (Extreme Programming), which are popular nowadays [Beck 00] For the development of object-oriented software systems, the Rational Unified Process [Jacobson 99] is discussed

All these models define a systematic way to achieve an orderly way of working during the project In most cases, phases and design steps are defined They have to be completed with a result in form of a document A phase completion, often termed as →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 are defined These tasks have to be accomplished by the project staff Sometimes, in addition to the models, the techniques and processes to be used in the particular phases are described With the aid of models, a detailed planning of the resources (time, personnel, infrastructure etc) can be performed In a project, the development models define for everybody involved the collective and mandatory definition of tasks to be accomplished and their chronological sequence

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

Waterfall-model
Testing as final inspection

The first fundamental model was the Waterfall-model (see figure 2-3, shown with the originally defined levels [Royce 70][4]) It is impressively simple and very well known Only when one development level is completed the next one will be initiated Only between adjacent levels there are feedback loops that allow, if necessary, required revisions in the previous level The crucial disadvantage 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" in analogy to a manufacturing inspection before handing over the product to the customer

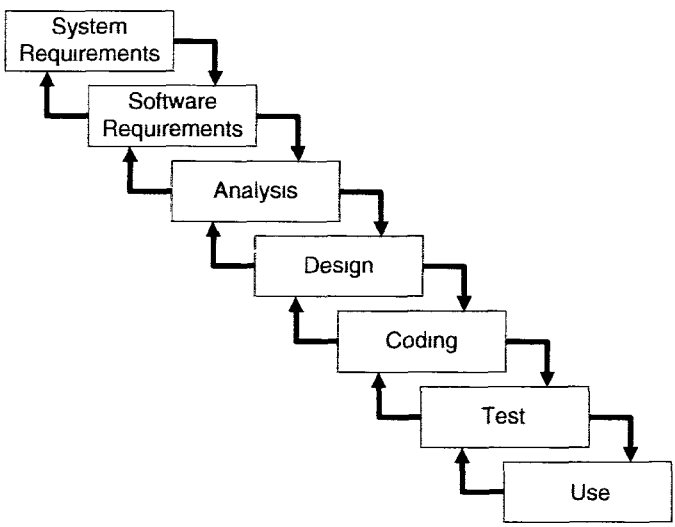


Fig. 2-3
Waterfall-model

General V model

An enhancement of the Waterfall-model is the general V-model ([Boehm 79], [IEEE/IEC 12207]), where the constructive activities are separated from the examination 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 ordered into test levels, matched to the appropriate abstraction level on the opposite side s constructive activity

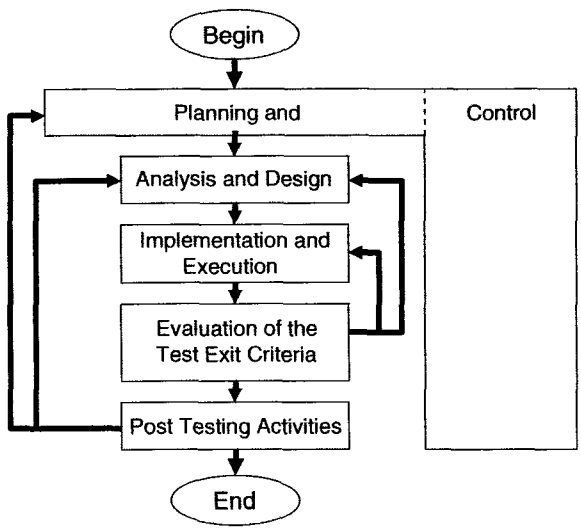
The general V-model is very common and is often used in practice

The description of tasks like they were discussed in the above mentioned process models is not sufficient as an instruction on how to perform structured tests in software projects. In addition to the embedding of 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. Namely into: →test planning and control,

4. Royce did not call his model Waterfall model, and he said in his paper "Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps "

test analysis and design, test implementation and execution, evaluation of test →exit criteria and reporting, and test closure activities. Although logically sequential, the activities in the test process may overlap or take place concurrently. These subtasks form a fundamental test process and are described in more detail in the following sections.

Fig. 2–4
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 start of the software development project. Like in every planning, during the course of the project the previous plans must be regularly checked, updated and adjusted.

Planning of the resources

The mission and objectives of testing must be defined and agreed upon. Necessary resources for the test process should be estimated. Which employees are needed for the execution of which tasks and when? How much time is needed and which equipment and utilities must be available? These questions and many more must be answered during planning. The result should be documented in the →test plan (see chapter 6). Necessary training of the employees should be provided. An organization structure with the appropriate test management must be arranged or adjusted if necessary.

Test control is monitoring the test activities and comparing what actually happens during the project with the plan, reporting status of deviations from the plan, and taking any actions to meet the mission

and objectives in the new situation. The test plan must be continuously updated taking into account the feedback from monitoring and control.

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 the appropriate reporting from the employees as well as data automatically generated from tools. Agreements on this point must be made early.

Determination of the test strategy

The main task of planning is to determine the →test strategy (see chapter 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 a higher attention, thus be tested more intensively. Less critical subsystems get less extensive testing. If no negative effects are expected in case of a failure, testing could even be skipped on some parts. But 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 of 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 possibility to place online-orders (within the first expansion stage by the dealer)
EasyFinance calculates an optimal way of financing for the customer
NoRisk provides the possibility of subscribing to an adequate insurance

Naturally, the 5 subsystems should not be tested with identical intensity. The result of a discussion with the ordering customer is that incorrect behavior of the subsystems *DreamCar* and *ContractBase* will have the most harmful effects. Because of this the test strategy decides that these two systems must be tested more intensively.

The possibility to place orders on-line, 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 (e.g. fax). But it is important that the order data must not be altered or get lost in the subsystem *JustInTime*. Thus, this aspect should be tested more intensively again.

For the other two subsystems *NoRisk* and *EasyFinance*, the test strategy defines that all of their main functions (computing a tariff, recording and placing contracts, saving and printing contracts etc.) have to be tested. Because of time constraints, it is not possible to cover all thinkable contract variants for the financing and insurance of a car. Thus, it is decided to con-

centrate the test around the most occurring tariff combinations. Combinations that are more seldom get a lower priority (see chapter 2.4 and 6.4)

These first thoughts about the test strategy for the VSR-System make clear that a definition of the intensity of testing is reasonable for whole subsystems as well as for single aspects of a system.

Define test intensity for subsystems and individual aspects

The intensity of testing depends very much on the used test techniques and the →test coverage that must be achieved. The test coverage serves as a test exit criterion. Besides →coverage criteria, which refer to the source code (for example statement coverage, see chapter 5.2), it is possible to define the fulfillment of the customer requirements as an exit criterion. It may be demanded that all functions must at least be tested once or for example that at least 70 % of the possible transactions in a system are executed once. Of course, the definitions of the exit criteria and thus the intensity of the tests should be done considering the risk. Here, all test exit criteria are defined, in order to be used after executing the test cases to decide if the test process can be finished.

Prioritization of the tests

Because software projects are often run under severe time pressure, it is reasonable to appropriately consider the time aspect in the 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 chapter 6.4).

Tool support

If the necessary tool support (see chapter 7) does not exist, selection and acquisition of tools must be initiated early. Existing tools must be evaluated for use in the actual situation. If parts of the test infrastructure have to be developed, this should be started immediately. →Test harnesses (or →test beds), where subsystems can be executed in isolation, must in most cases be programmed by the developing or testing organization. They must be created in time, to be ready after programming the respective test objects. If frameworks are applied, like Junit [URL: xunit], their usage in the project must be checked and announced early.

2.2.2 Test Analysis and Design

The first thing to do is to review the →test basis, i.e. the specification of what should be tested. The specification may not be concrete or clear enough to develop test cases. E.g. a requirement is so imprecise in defining the expected output or the expected behavior of the system, that no test cases can be specified. The →testability of the requirement is insufficient. Rework of the requirements has to be done. Determin-

ing 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.

The test strategy determined in the test plan defines which test techniques shall be used. The test strategy may be further detailed in this step. However, the most important task is to develop test cases. Appropriate test cases are then developed using the test techniques specified in the test plan, as well as techniques chosen based on an analysis of possible complexity in the test target.

Logical and concrete test cases

The 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 that the actual inputs are chosen (→concrete test cases). Also, the opposite way is possible: from the concrete to the general logical test case. This procedure must be used if a test object is specified insufficiently and test specification is done in a rather experimental way (→exploratory testing, see chapter 5.3). The development of physical test cases, however, is part of the next phase, test implementation.

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

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

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 test oracle. Here are two possibilities

The tester derives the expected data from the input data by calculation or analysis based on the specification of the test object.

If functions that do the opposite action are available, they can be run after the test and it is then checked if the result is the old input. An example is encryption and decryption.

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:

Test cases for examining the specified behavior, output and reaction. Included here are test cases that examine the specified handling of exceptional and error cases. But it is often difficult to create the necessary conditions for the execution of these test cases (e.g. capacity overload of a network connection).

Test cases for examining the reaction of test objects to invalid and unexpected inputs or conditions, which have no specified →exception handling.

Examples of test cases

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

A company orders an application that is supposed to calculate the Christmas bonus of the employees depending on the length of their company affiliation. In the description of the requirements the following is found: "Employees with a company affiliation of more than three years, get 50 % of their monthly salary as Christmas bonus. Employees that have been working more than five years in the company get 75 %. With an affiliation greater than eight years, a 100 % bonus will be given.

This text shows the following cases for the bonus depending on the affiliation:

Company affiliation <= 3	results in a bonus = 0 %
3 < company affiliation <= 5	results in a bonus = 50 %
5 < company affiliation <= 8	results in a bonus = 75 %
Company affiliation > 8	results in a bonus = 100 %

Based on this it is possible to create the following logical test cases (see table 2-1).

Tab. 2-1
Table with logical test cases

Test case number	Input x (company affiliation)	Expected result (Bonus in %)
1	X <= 3	0
2	3 < x <= 5	50
3	5 < x <=8	75
4	X > 8	100

To execute the test cases, the logical test cases must be converted into concrete test cases, i.e. concrete input data must be defined (see table 2-2). Special initial situations or conditions are not given for these test cases.

Test case number	Input x (company affiliation)	Expected result (Bonus in %)
1	2	0
2	4	50
3	7	75
4	12	100

Tab. 2-2
Table with concrete test cases

The chosen inputs in these examples are merely supposed to show the differences between the logical and concrete test cases. Any explicit test technique to create these test cases was not used. Furthermore, we do not claim that the four test cases adequately examine the bonus calculation. For example, there are no test cases for the treatment of invalid inputs (e.g. a negative company affiliation). More detailed information about creating systematic test cases with appropriate test techniques can be found in chapter 5.

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

2.2.3 Test Implementation and Execution[5]

Test implementation and execution is the activity where test conditions and logical test cases are transformed into concrete test cases, all the details of the environment are set up to support the test execution activity, and the tests are executed and logged.

When the testing process has advanced and more is known about the technical implementation, the concrete, physical test cases will be developed from the logical ones. These test cases can then be used without further modifications or additions for executing the test. To state this in more detail: The preconditions and input values will then be used.

Test case execution

In addition to defining test cases one must describe how the tests will be executed. The priority of the test cases (see chapter 6.2)

5. In other literature, test implementation is often called test preparation, and test execution is often defined as a work phase by itself.

decided during test planning, must be taken into account. If the test developer executes the tests, any more detailed description may not be necessary.

The test cases should also be grouped into test suites for efficient test execution and easier overview.

Test harness

In many cases specific →test harnesses, →drivers, →simulators etc. must be programmed, built, acquired or set up as part of the test implementation. Because failures may also be caused by faults in the test harness, the correct functioning of the test environment must be checked.

When all preparing 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

It is recommended to start test execution with the examination of the test object's main functionality. If failures or →deviations from the expected result already show up at this time, it is foolish to test more deeply. The wrong main functions should be corrected first. After passing this test, everything else is tested. Such a proceeding should be defined in the test strategy.

Tests without a protocol are of no value

The test execution must be exactly and completely logged. Logging what testing activities have been carried out, i.e. logging every test case run and its success or failure, or logging its results for later analysis. On one hand, the test execution must be comprehensible to not directly involved people, for example to the customer, on the basis of these →test logs. On the other hand, it must be provable that the planned test strategy was actually executed. The test log must document who tested which parts when, how intensively and with which results.

Reproducibility is important

Besides the test object, quite a number of documents and information belong to each test execution: test environment, input data, test logs etc. The information belonging 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. In some cases it must also be possible to audit the test. The testware must be taken under configuration management (see also chapter 6.7).

Failure found?

If, during test execution, a difference shows up between expected and actual results, it must be decided when evaluating the test logs, if it really is a failure. If this is the case, the failure must be documented

and a first rough analysis of possible causes must be made. This may require to specify and execute additional test cases.

The cause can be an erroneous or inexact test specification, problems with the test infrastructure or the test case or wrong test execution. It must be examined carefully which of all these possibilities applies. Nothing is more detrimental to the credibility of a tester than a reported assumed failure, whose cause actually is a test problem. But the fear of this to happen should not result in potential failures not being reported, i.e. the testers starting to self-censor their results. This could be fatal in almost the same manner.

In addition to reporting discrepancies, test coverage should be measured (see chapter 2.2.4) and if necessary the use of time should be logged. Tools for this should be used (see chapter 7).

Correction may lead to new faults

Invoking →incident management: Based on the →severity of a failure (see chapter 6.6.3) the priority of fault correction must be decided. After the correction, it must be examined if the fault has really been corrected and no new faults have been introduced (see chapter 3.7.4). New testing activities result from the action taken for each incident, e.g. 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, in order to avoid unwanted interactions of the changes. In practice this is not often possible due to the extra cost for every installation of new software into the test environment. To report every failure individually to the developer and to continue testing only after the fault is corrected leads to not justifiable effort. Several faults should be corrected as a group and then the program should be resubmitted to testing with a new version state.

The most important test cases first

In many projects, there is not enough time to execute all specified test cases. Then a reasonable selection of test cases must be made to make sure that as many critical failures as possible are detected even with a limited number of executed test cases, or that the most critical parts of the system under test are demonstrated. Therefore, test cases should be prioritized. In case of a premature stop of the tests, the best possible result should be achieved. This is called →risk-based testing (see chapter 6.4.3).

Furthermore, giving priority has the advantage 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 then be tested insufficiently and resources would be wasted on uncritical parts for no reason.

2.2.4 Evaluation of the Test Exit Criteria and Reporting

End of test?

This is the activity where the product is assessed against set objectives, the test exit criteria[6] specified earlier. 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 had a too high level.

It must be decided if the test exit criteria defined in the test plan are fulfilled. Considering the risk, an adequate exit criterion must be determined for each used test technique. For example, it could be specified that a test is sufficient if 80 % of the statements of the test object were executed during test execution. But this would not be a strong test criterion. Appropriate tools should be used to collect such measures or →metrics in order to decide the end of the test (see chapter 7.1.4).

If at least one test exit criterion is not fulfilled after executing all tests, further tests must be executed. Attention should be paid to make sure that the new test cases lead to fulfilling the respective exit criteria. Otherwise, the extra test cases just result in additional work, but no improvement concerning the test end.

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. Further tests are then canceled. Such a decision must, naturally, consider the associated risk.

An example for 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 exceptional situation. The appropriate source code for treating the exceptional situation can then not be executed and tested. In such cases, other examination procedures should be used. For example static analysis (see chapter 4.2).

Dead code

A further case of non-fulfillment of test exit criteria may occur, if the specified criterion is impossible to fulfill in the concrete case. If for example, if the test object contains →dead code, this cannot be executed. Thus, 100 % statement coverage is not possible, as this would also include the unreachable (dead) code. Even this possibility must be considered in order to avoid further senseless tests in relation to the criterion.

An impossible criterion is often a hint to possible inconsistency in or impreciseness of the requirements or specifications. In the given

6. Also called test completion criteria.

example, it makes certainly sense to investigate why the program contains non-executable instructions. This way, further faults can be found or their effects can be avoided in advance.

If further tests are planned, the test process must be resumed and it must be decided on which point to re-enter it. 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 amended in order to fulfill the required exit criterion.

Further criteria for the determination of the test's end

In addition to test coverage criteria, other criteria can be used to define the test end. Another possible criterion is the failure rate or the →defect detection percentage (DDP). Figure 2-5 shows the average number of new failures per testing hour over ten weeks. In the first week, there was an average of two new failures per testing hour. In the tenth week, it is less than one failure per two hours. If the failure rate falls below a given threshold (e.g. less than one failure per testing hour), it will be assumed that more testing does not pay off anymore and the test can be ended.

Doing this it must be considered that failures can have very different effects. A classification and differentiation of failures, according to the impact to the stakeholders, i.e. failure severity, is therefore reasonable and should generally be considered (see chapter 6.6.3).

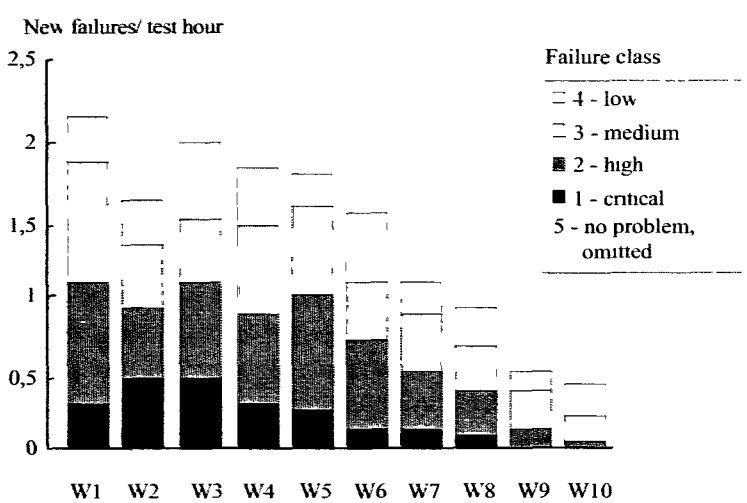


Fig. 2-5
Failure rate

Consider several test cycles

The failures found during the test must be repaired according to their severity and a new test becomes necessary. Cycles often result if further failures occur during the test of modifications. These need to be isolated and corrected and a new test cycle is necessary. Not planning

such correction cycles, assuming that no failures will occur while testing, is unrealistic, because it can be assumed that testing finds failures, whose corresponding faults must be removed and which must be tested in a further →test cycle. Project delays are then standard. 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 has to provide for the appropriate time buffers and personnel resources.

Exit criteria in practice: Time and costs

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 more resources than planned were used in testing, 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 chapter 6.3.1).

Test summary report

When the test criteria are fulfilled or their non-fulfillment is clarified, a →test summary report should be written for the stakeholders. In lower level tests, this may just take the form of a message to the project manager about meeting the exit criteria. In higher-level tests, there may be a formal report.

2.2.5 Test Closure Activities

The activities, which should be executed during this final phase in the test process, are often left out. The experiences made during the test work should be analyzed and made available for further projects. Interesting are deviations between planning and execution for the different activities as well as 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 are achieved – if at all?
Which unexpected events happened (reasons and how they were circumvented)?
Are there open →change requests? Why were they not implemented?
How good 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 spent resources and the achieved results, will probably show possibilities for improvement. If these findings are used in the next 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 finishing activity is the "conservation" of the testware for the future. Software systems are used for a longer period. During this time, failures not found during testing will occur. Additionally, the customers require changes. Both lead to new versions of the program, and the changed program must be tested in any case. A major part of this test effort during →maintenance can be saved if the testware (test cases, test logs, test infrastructure, tools, etc.) are still available. The testware should be delivered to the organization responsible for maintenance. The testware can then be adapted instead of being constructed from scratch and can also be successfully used for projects with similar requirements, after adaptation.

2.3 The Psychology of Testing

Errare humanum est

People make mistakes, but they do not like to admit them! One goal of testing software is uncovering discrepancies between the software and the specifications or customer needs. The failures found must be reported to the developers. This chapter describes how psychological problems can be handled.

The tasks of developing software are often seen as constructive actions. The tasks of examining documents and software are seen as destructive actions. Just out of this perception there already differences in the involved people's attitude to their job. But these differences are not justifiable, because "testing is an extremely creative and intellectual challenging task" [Myers 79, p.15].

Developer test

"Can the developer test his own program?" is an important and frequently asked question. A universally valid answer does not exist. If the tester is also the author of the program, she must examine her own work very critically. Only a few people are able to keep the necessary distance to the self-created product. Who really likes to prove his own errors? There is rather the interest to show that the own source code works well.

The big weakness of developer tests is that every developer who has to test his or her own program parts will tend to be too optimistic.

There is the danger of forgetting reasonable test cases because the developer is more interested in programming than in testing or only tests superficially.

Blindness to one's own errors

If the developer implements a fundamental design error, for example if she misunderstood the conceptual formulation, it is not possible that she finds this using her own tests. The proper test case will not even come to mind. One possibility to decrease this problem of "blindness to the own errors" is to work together in pairs and let the programs be tested by a colleague.

On the other hand, it is an advantage to have a good knowledge of one's own test object. It is not necessary to learn the test object and thus time is saved. Management has to decide when it is an advantage to save time even with the disadvantage of blindness for own errors. The decision must be made depending on the criticality of the test object and the associated failure risk.

Independent testing team

An independent testing team tends to increase the quality and comprehensiveness of the tests. The tester can look at the test object without bias. It is not "her" product and possible assumptions and misunderstandings of the developer are not necessarily the assumptions and misunderstandings of the tester. The tester must acquire the necessary knowledge of the test object in order to create test cases. This costs time. But the tester comes along with a deeper test-know-how, which a developer does not have or must acquire at first (or rather should have acquired before, because the necessary time is often not available).

Reporting of failures

It is the job of the tester to report the failures and discrepancies observed to the author and/or to the management. The way of doing this can contribute to the cooperation between developer and tester or have a negative influence on the important communication of these two groups. To prove other peoples' errors is not an easy job and requires diplomacy and tact.

There is often the problem that failures found during testing are not reproducible for the developers in the development environment. Besides the detailed description of failures, the test environment must also be documented in detail in order to be able to trace any differences in the environments, 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 if this is a fault or not

does not help. Also the often heard reaction of developers against any comment: "It's not a bug, it's a feature!" is not very helpful.

Mutual comprehension

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

The illustrated 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 has then the task to decide whether there still is a chance to meet the deadline and possibly deliver software with known problems or if the delivery must be delayed and additional time should be used for corrections. This decision must be made depending 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. Appropriate 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 not possible.

An exhaustive test where all inputs values and their combinations are run, combined with taking into account all different →preconditions, is impossible. Software appearing in practice would require an "astronomically" high number of test cases. Because of this, every test is always 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 lifecycle and focus on defined goals. This contributes to finding defects early.

Principle 4:

Defects tend to cluster together.

Often, most defects are found in few parts of the test object. Defects are not evenly distributed, but cluster together. 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

If the same tests are repeated over and over again, they tend to loose effectiveness. New, until now unknown, defects are not found any more. Thus, in order to maintain the effectiveness of tests and to fight this "pesticide paradox", new and modified test cases should be developed. Parts of the software that until now were not tested, or not yet used input combinations will then be executed and more defects may be found.

Principle 6:

Test 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. For every software system, the test exit criteria etc. should be decided individually depending on its usage environment. Safety critical systems require different tests than e-commerce applications.

Principle 7:

The fallacy of assuming that no failures means a useful system.

Finding failures and repairing defects does not guarantee that the system as a whole 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 problems.

2.5 Summary

Technical terms in the domain of software testing are often defined and used very differently. This can result in misunderstandings. The knowledge of standards (e.g. [BS 7925-1], [IEEE 610.12], [ISO 9126]) and terminology defined in them is therefore an important part of the education to the Certified Tester. The glossary in the appendix of this book 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 phases planning and control, analysis and design, implementation and execution, evaluation of the test exit criteria and reporting, and test closure activities. The test can be finished when the 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 behavior. If the expected result and actual result are different, there is a failure. The expected results should be defined before test execution 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.

3 Testing in the Software Lifecycle

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 methods that are used during the development process.

Each project in software development should be approached using a life cycle model (see [IEEE/IEC 12207]) that has been chosen in advance. Some important models have been presented and explained in chapter 2.2. Each one of these models implies certain views on software testing. From the point of view of testing, the general V-model according to [Boehm 79] plays an especially important role here.

The role of testing within lifecycle models

In Boehm's model, the testing activities play an equally important role to development and programming. This has had a lasting influence on the appreciation of software testing. Every tester, but also every developer should know this general V-model and the views on testing implied there. Even if a different development model is used in a project, the principles presented in the following sections can be transferred and applied.

3.1 The General V-model

The main idea behind the general V-model is that development tasks and testing tasks are equally important and corresponding activities. The two branches of the letter "V" symbolize this.

The left branch represents the development process. Here, the system is gradually being designed and as last step being programmed. The right branch represents the integration and testing process, where the program elements are successively being assembled to form bigger, subsystems (integration), and where their functionality is tested.

→Integration and Test end with the acceptance test of the entire system developed according to this process.

Figure 3-1 shows such a V-model.[7]

7 The V-model is used in many different versions. The name and the number of levels vary according to literature source or the interpretation of the enterprise that uses it.

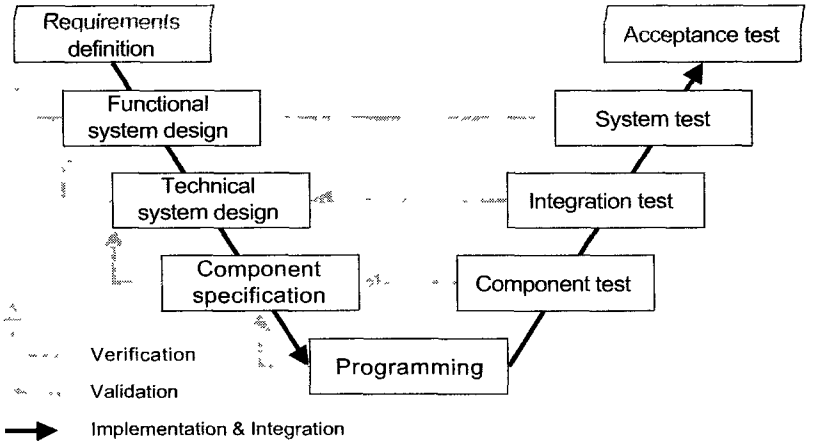


Fig. 3-1
General V-model

The constructive activities of the left branch are the ones we know from the Waterfall-model:

Requirements specification: The needs and requirements of the customer or the future system user are gathered, specified and finally approved. Thus, the purpose of the system and the desired characteristics and features are defined.
Functional system design: The requirements are mapped onto functions and dialogues of the new system.
Technical system design: The implementation of the system is designed. This includes the definition of interfaces to the system environment and the decomposition of the system into smaller understandable subsystems (system architecture). Each subsystems can then be developed independently.
Component specification: for each subsystem, its task, behavior, inner structure and its interfaces to other subsystems are defined.
Programming: Each specified component (module, unit, class) is implemented in a programming language.

Along these construction levels the software system is described in more and more detail. If mistakes are made during this construction, usually, the easiest way to find them is to look for them on the level of abstraction on which they are produced. Thus, the right branch of the V-model defines a corresponding test level for each construction level:

Component test (see chapter 3.2) verifies whether each software →component performs correctly according to its specification.

→**Integration test** (see chapter 3.3) checks if groups of components collaborate in the way that is specified by the technical system design.
System test (see chapter 3.4) verifies whether the system as a whole meets the specified requirements.
→**Acceptance test** (see chapter 3.5) checks if the system meets the requirements as specified in the contract, from the customers point of view.

Within each test level, it must be checked whether the outcomes of development meet the requirements specified or relevant on this specific level of abstraction. This process of checking the development results according to their original requirements, is called →validation.

Are we building the right system?

During validation[8] the tester judges whether a product (or a part of it) solves its task and therefore if this product is suitable for its intended use.

In addition to validation testing, the V-model also requires verification[9] testing.

Unlike validation, →verification refers to only one single phase of the development process. Verification shall assure that the outcome of a particular development phase has been built correctly and completely, according to its specification (the input documents for that development level).

Are we building the system right?

That means, it is examined whether specifications are correctly implemented, whether the product meets its specification, but not whether the resulting product is suitable for its intended use.

In reality, every test includes both aspects; but the validation aspect increases from lower to higher levels of testing. 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 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 the systems implementation. This is wrong. The tests on the right hand side of the model shall be interpreted as the levels of test execu-

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

tion. Test preparation (test planning and control, test analysis and design) starts earlier and is performed in parallel to the development phases on the left branch[10].

The differentiation of test levels in the V-model is more than only a temporal subdivision of testing activities. It is rather a process of defining abstraction levels in testing. These levels are technically very different, have different objectives and thus imply different methods, tools and require personnel with different knowledge.

The exact contents and the process for each test level are explained in the following chapters.

3.2 Component Test

3.2.1 Explanation of Terms

Within the first test level, component testing, the software units that have been implemented just before during the programming phase are tested systematically for the first time.

Depending on the programming language the developers have used, these software units are called differently, e.g. modules, units, programs, or functions. In object-oriented programming, they are called classes. The respective tests therefore are called →module, →unit, program, or →class tests.

Component and component test

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

3.2.2 Test Objects

The main attribute of component testing is the following: 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 test checks aspects internal to the component

The component under test can also be assembled from various smaller components. However, it is crucial that the tester focuses on internal aspects and behavior of the component. Component testing is not about testing the interaction with neighbor components. This is the task of integration testing.

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

3.2.3 Test Environment

Component test as lowest test level deals with test objects coming "directly from the developers desk". It is obvious that in this test level testing is performed in tight cooperation with development.

Example: Testing of a class method

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

> The starting point is `baseprice` minus `discount`, where `baseprice` is the basic price of the vehicle and `discount` is the discount to this price granted by the dealer.
> A `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 only. If five or more special equipment items are chosen, this discount is increased to 15 percent.
> The discount that is granted by the dealer applies only on the `baseprice`, whereas the discount on special items applies to the special items only. Both discounts cannot be added.

The following C++-function calculates the total price[11]:

```
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;
```

11. Actually, there is a defect in this program: Discount calculation for more than 5 is never reachable. The defect is used when explaining the use of white box analysis in chapter 5.

In order to test the price calculation, the tester uses the corresponding class interface; she calls the function `calculate_price()` with appropriate parameters and data. Then she 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 (e.g. `calculate_price()`) and then receives the test objects 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);[12]

// 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 above 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 file or a database.

In order to write test drivers, programming skills and knowledge of the component under test is necessary. The test object's program code (in the example, a class function) must be available and understood by the tester, so that the call of the test object can be correctly programmed in the test driver. To write a suitable test driver the tester has to know the programming language and suitable programming tools must be at hand.

12. 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.

This is why mostly the developers themselves do the component testing. Although a component test is meant, this is then often also called developer test. The disadvantages occurring if a programmer is testing his own program have already been discussed in chapter 2.3.

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

Hint

The use of generic test drivers can help to simplify the component test, because the costly programming of various test drivers for each single component is not necessary anymore. Generic test drivers are available on the market (e.g. [URL: xunit]) or are to be produced project specific. If a generic test drivers is used, testing by team colleagues[13] that are not familiar with the particular component and the programming environment is easier. The test driver should, for example, provide a command interface and comfortable mechanisms for handling the test data and for recording and analyzing the tests. All test data and test protocols will be structured in a very similar way. An analysis of the tests across several components is then possible.

3.2.4 Test Objectives

The test level 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 guarantee that the particular test object executes its entire functionality correctly and completely, as required by the specification (see →functional testing).

Here, functionality means the Input/Output-behavior of the test object. In order 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).

13. Sometimes, two programmers work together, every one of them testing the components that the colleague has developed. This is called "buddy testing" or "code swaps". Some "pair programming" usage has this meaning, too.

Example: Testing the VSR price calculation

The test cases for the price calculation of the "CarConfigurator" in the previous example show very clearly, how the examination of the Input/Output-behavior works. Each test case calls the test object with a particular combination of data, that is the price for the vehicle in combination with a different set of extra equipment items. Then it is examined whether the test object, given these input data, calculates the correct price.

Test case 2 for example checks the partial functionality "discount in the case of five or more special equipment items". If test case 2 is executed, we can see that the test object calculates a wrong total price. Test case 2 produces a failure. The test object does not completely fulfill the requirements as stated by its specification.

Typical software defects found during functional component testing are wrong calculations, 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 has to cooperate with many neighbor components and exchange data with them. It is not possible to exclude the possibility that a component will be called (or used) in a way not according to its specification. I.e., the component is misused. In such cases, the component that is called should not suspend its service or cause the whole system to crash. It should rather be able to handle the error situation in a reasonable and robust way.

Testing robustness

This is why testing for →robustness is another very important aspect of the component test. The way to do this resembles the functional tests. However, function calls and test data are used that are either wrong or at least special cases not mentioned in the specification. 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 through a null pointer. Such faults could lead to a program crash.

Example: Negative Test

In the price calculation example, such negative tests are for example function calls with negative or far too large values or wrong data types ("`char`" instead of "`int`" etc.)[14]:

14 Depending on the compiler, data type errors can already be detected during the compiling process

```
// 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);
```

Some interesting aspects become clear: ***Excursion***

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 a program's code deals with exception handling. Robustness has its cost.

Component test 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 expense) should be checked during component test. This for example refers to non-functional characteristics like efficiency and maintainability.

Efficiency states how efficient the component uses computer resources. Here we have different aspects like use of memory, computing time, disk or network access time and the time needed to execute the component's functions and algorithms. In contrast to most other nonfunctional tests, a test object's efficiency can be measured exactly during the test. This is done by exactly measuring suitable criteria (for example memory usage in kilobytes, response times in milliseconds). Efficiency tests are seldom done for all the components of a system. Efficiency must be verified only in efficiency-critical parts of the system or if efficiency requirements are stated by specifications. This happens, for example, in testing embedded software, where only limited hardware resources are available. Another example is the case of real-time systems where given timing constraints must be guaranteed. *Efficiency test*

Maintainability means all the characteristics of a program that have an influence on how easy or how difficult it is to change the program or continue developing it. Here, it is crucial how much effort the developer needs to understand the program and its context. This counts for the developer of the original program who is asked to continue development after months or years as well as for a programmer who takes over responsibility for the code that someone else has written. Thus, the following aspects are most important for testing maintainability: Code structure, modularity, and quality of the comments *Maintainability test*

in the code, adherence to standards, understandability and currency of the documentation etc.

Example: Code that is hard to maintain

The code that is shown in the example `calculate_price()` shows some deficits. There are no comments, numeric constants are not defined but just written into the code. If such a value must be changed afterwards, it is not clear, whether and where this value occurs in other parts of the system and 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 test and especially reviews (see chapter 4.1) are the right means for that purpose. Because the characteristics of a single component are examined, it is best to include such analyses in the component test.

3.2.5 Test Strategy

As we explained before, component test is very close to development. The tester usually has access to the source code, which makes component testing the domain of white box testing (see chapter 5.2).

White box test

The tester can design test cases using her knowledge about the component's program structures, its functions, and variables. Access to the program code can also be helpful for executing the tests. With the help of special tools (→debugger, see chapter 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 can be observed but also be manipulated with the debugger. This is especially useful for robustness tests, because the tester is able to trigger special exceptional situations.

Example: Code as basis for testing

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 whatsoever on this situation; the implemented functionality is not supposed to be there.

In reality however, component test is often done as a pure "black box" test, this means that the inner structure of the code is not used to design test cases[15]. On the one hand, real software systems consist of hundreds or thousands of elementary components. To analyze the code for designing test cases is probably only feasible with very few, selected components. On the other hand the elementary components will later be integrated to larger units. Often, the tester can only see 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. When planning integration and testing, therefore the question has to be answered whether to test elementary parts or only larger units in the component test.

Test-driven development

A modern approach in component testing, popular in incremental development, is to prepare and automate test cases before coding. This is called a →test-first programming or test-driven development. This approach is highly iterative. Pieces of code are tested and then improved until the component passes all it's tests (see [Link 03]).

3.3 Integration Test

3.3.1 Explanation of Terms

After the component test, the second test level in the V-model is the integration test. Integration testing supposes that the test objects subjected to it (that means components) have already been tested. Defects should, if possible, already be corrected.

Integration

Groups of these components are composed to form larger structural units and subsystems. This connecting of components is called integration and is done by developers, testers or special integration teams.

Integration test

After assembling the components, it must be tested if all components collaborate correctly. The goal of this integration testing therefore is to expose faults in the interfaces and in the interaction between integrated components.

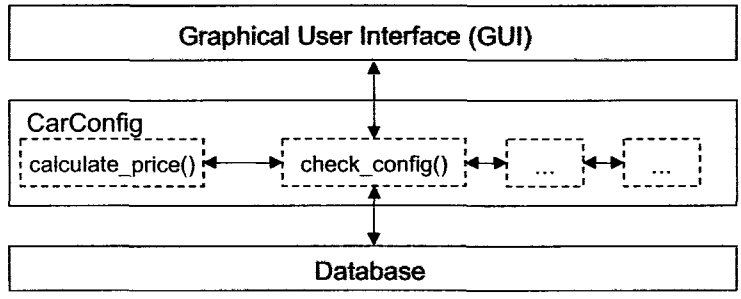
Why is integration testing necessary, if every single component has already been tested? The following example illustrates the problem:

15. That is a big mistake, because that leaves untested a sizeable percentage of the code – often as much as 60–80 %. This untested code is, of course, a perfect hiding place for bugs.

Example: Integration test VSR-DreamCar

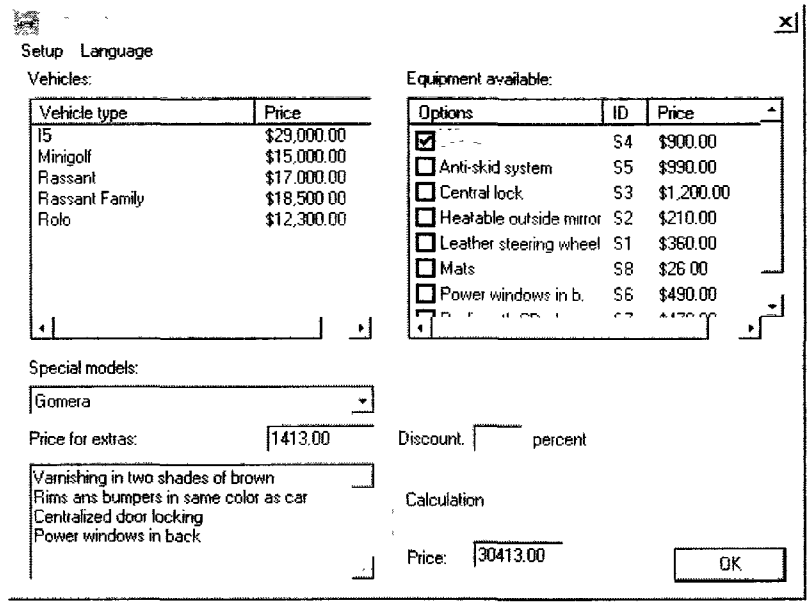
The VSR subsystem *DreamCar* (see figure 2-1) consists of several elementary components.

Fig. 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 in figure 3-3.

Fig. 3-3
User Interface VSR-DreamCar



When the user has chosen her 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 from the vehicle that was chosen,

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, one can choose accessories (e.g. alloy rims) without paying for them. Somewhere on the way from the GUI to `calculate_price()` the calculation misses the fact that alloy rims were chosen.

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 is executed before, such interface problems can still occur. That is why integration test is necessary as a further test level. Its task is to find collaboration and interoperability problems and isolate their causes.

Example: VSR integration test

The 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 *JustIn-Time* (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 computing center of the enterprise.

Integration testing in the large

As the example shows, interfaces to the system environment, i.e. external systems, are also subject to integration and integration testing. If interfaces to external software systems are examined, we sometimes speak of a →"system integration test", "higher level integration test" or "integration test in the large" (integration of components is then "integration test in the small", sometimes called →"component integration test"). The fact that the development team has only "one half" of such an interface under its control 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. A passed integration test is no guarantee for flawless functioning at this point for all future time.

3.3.2 Test Objects

Assembled components

In the course of integration, the single components are assembled step by step to larger units (see chapter 3.3.5). Ideally, there should be an integration test after each one of these steps. Every subsystem gener-

ated like this can afterwards be the base for the integration of even larger units. Such composed units might itself again be objects of the integration test.

External systems or off-the-shelf products

In reality, a software system is seldom developed from scratch. An existing system is changed, extended or linked to other systems. Also, many system components are commercial →off-the-shelf software products (COTS) that are bought from the market (for example the database in *DreamCar*). In the component test, 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.

3.3.3 Test Environment

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

Re-use of the testing environment

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

In a badly organized component test, there may only be test drivers for some few components. Their user interface may also be totally different. This will create trouble: The tester now (during integration testing in a much later stage of the project) has to put much effort into the creation, change or repair of the test environment. This means that valuable time needed for testing is lost.

Need for monitors

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

3.3.4 Test Objectives

Wrong interface formats

The test objective of the test level "integration test" is obvious: to reveal interface and cooperation problems as well as conflicts between integrated parts. Problems can already arise even when an attempt is

made to integrate two single components. Linking them together might not work because, for example, their interface formats are not compatible to each other, or because some files are missing or because the developers have split the system in completely different components than designed (see static testing, chapter 4.2).

Typical faults in data exchange

The harder to find problems, though, are due to the execution of the connected program parts. These can only be found by dynamic testing. These are faults in the data exchange or communication between the components. The following types of faults can roughly be distinguished:

A component transmits syntactically wrong 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 in a different way (functional fault of a component, contradicting or misinterpreted specifications).
Data are transmitted correctly but they are transmitted at the wrong time or late (timing-problem) or the intervals between the transmissions are too short (throughput, load or capacity problem).

Example: Integration problems in VSR

According to above mentioned failure types the following interface failures could occur during the VSR integration test:

In the GUI of the *DreamCar*, selected extra equipment items are not passed on to `check_config()`. Therefore, the price and the data of the order 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, the code numbers are interpreted differently (there, 442 probably represents red). A correct order from the VSR would lead to delivery of a 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" supposes 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 could be found in the component test, because the resulting failures only occur when interaction between two software components happens.

In addition to testing functionality, also non-functional tests may be executed in integration testing if such attributes are important or

are considered at risk. These may include testing performance and capacity of interfaces.

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, it is possible to do that and we can see this done in reality. But this may result in 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 (see chapter 3.2.4).

The cost of trying to save effort by cutting the component test is finding fewer faults and having more difficulty in diagnosis, thus spending more, rather then less effort. The combination of a component test and a subsequent integration test is most often more effective and efficient.

3.3.5 Integration Strategies

In which order should the components be integrated in order to execute the necessary testing as quickly and easily as possible? How do we get the greatest possible efficiency of testing? Efficiency is the relation between the cost of testing (expense in testing personnel and the usage of tools etc) and the benefit of testing (number and severity of the problems revealed). It is the test managers task to figure this out and to choose and implement an optimal integration strategy for the project.

Components are completed at different times

In practice, there is the difficulty that the different software components are completed at different times. These can be weeks or even months apart. No project manager and no test manager can tolerate that the testers have to wait for so long time doing nothing until the development of all the components is finished and they are ready to be integrated.

An obvious ad-hoc strategy to solve this problem fast is to integrate in the order in which the components are ready. This means: As soon as a component has passed the component test, it is checked whether it fits to 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. However, integration test planning should attempt to organize the delivery of components to correspond risk, system architecture etc.

Example: Strategy of integration in the VSR project

In the project VSR, the central subsystem *ContractBase* turns out to be more complex than expected. The completion of it is delayed for several weeks because the work on it is much more costly than originally expected. In order not to lose 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*. In order to calculate the price of the insurance, *NoRisk* needs to know which type of vehicle was chosen, because this determines the parameters of the insurance. As a temporary replacement for *ContractBase*, a →stub is programmed. This stub receives simple car configuration data from *DreamCar*, then determines the vehicle type code from these data and passes it on to *NoRisk*. Furthermore, the stub makes it possible to put in different relevant data about the customer. *NoRisk* calculates the insurance price from these data and indicates it in a window for checking. The price and other data are then saved in a test log. The stub serves as provisional replacement for the yet missing subsystem *ContractBase*.

This example makes clear: the earlier the integration test is started (in order to save time), the more effort is necessary for programming of stubs. The test manager has to choose her integration strategy optimizing both factors (time saving vs. cost for the testing environment).

Constraints for integration

Which strategy is optimal (most time saving and least costly) depends on the individual circumstances in each project. These 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 single parts of the system are developed and when they should be ready for testing. However, when determining the order of implementation the tester or test manager should be consulted.

The **test plan** determines which aspects of the system are to be tested, how intensely this is done 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 a viable integration strategy. As the integration strategy depends on delivery dates, the test manager should consult the project

manager in writing the project plan. The order of implementation of the components should be suitable for integration testing.

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

Basic 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 main part of the test environment.

Disadvantage: Lower level components not yet integrated must be replaced by stubs. 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 these integrated parts are tested.

Advantage: No stubs are needed.

Disadvantage: Higher-level components must be simulated by test drivers.

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 strategy

A skeleton or backbone is built into which components are gradually integrated [Beizer 90].

Advantage: Components can be integrated in any order.

Disadvantage: Labor intensive skeleton or backbone is required.

Top-down or Bottom-up integration in their pure form can only be applied to program systems that are structured in a strictly hierarchical way; in reality, this occurs not often. This is why in reality, a more or less individual mixture of above mentioned integration strategies[16] is chosen.

Avoid the big bang!

Any non-incremental integration – also called →"big bang" – should be avoided. Big bang integration means waiting with the integration until all software elements are developed and then throwing everything together in one step. This typically happens due to lack of any integration strategy. In the worst case even component testing is skipped. The disadvantages of this are obvious:

The time until the big bang is lost time that could have been spent for testing. As testing always suffers from lack of time, not a single day that could be used for testing should be wasted.
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 next test level is the System test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after the component and integration tests? The reasons for this are:

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[17]. The testers validate whether the requirements are met completely and appropriately.
Many functions and system characteristics result from the interaction of all system components and consequently, they are only visible on the level of the entire system and can only be observed and tested there.

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

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

Example: VSR-System tests

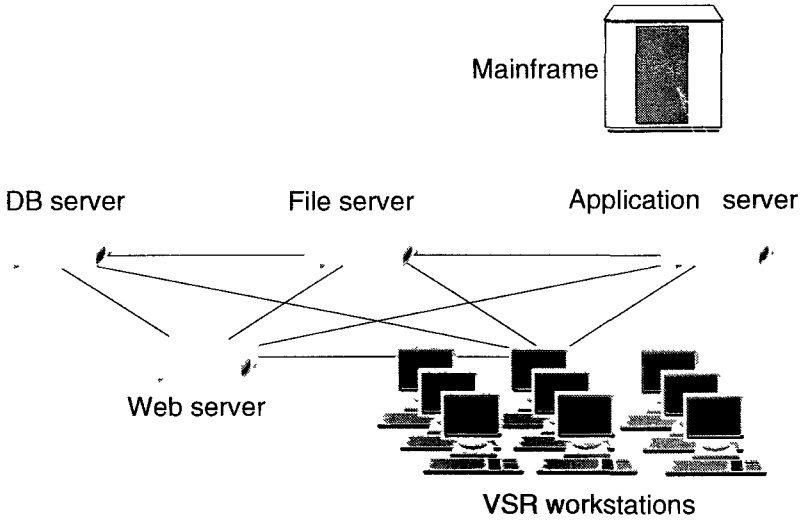
The main purpose of the VSR-System is to make ordering a car as easy as possible. Ordering a car the user uses all the components of a VSR-System: The car is configured (*DreamCar*); financing and insurance are calculated (*EasyFinance*, *NoRisk*); the order is transmitted to production (*JustInTime*) and the contracts are archived (*ContactBase*). The system suits its purpose only when all of these system functions and all of the components collaborate correctly. The system test checks whether this is the case.

3.4.2 Test Object and Test Environment

After the completion of the integration test, the software system is completely assembled and the system test looks at the system as a whole. This is done in an environment as similar as possible to the later →operational environment.

Instead of test drivers and stubs, the hardware and software products that are 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 a VSR-System test environment.

Fig. 3–4
Example of a system test environment



The system test requires a separate test environment

In order to save costs and effort, one mistake is often made: Instead of testing the system in a separate environment, the system test is run in the customer's operational environment. This is harmful because of the following reasons:

During system testing there will of course occur failures. And damage to the customer's operational environment can result. Expensive system crashes and data loss in the production system can occur.
The testers have no or only limited control over parameter settings and configuration of the operational environment. The test conditions can gradually change because the other systems in the customer's environment are running simultaneously to the test. The system tests that have been executed cannot be reproduced or can only be reproduced with difficulty (see chapter 3.7.4).

System test effort is often underestimated

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

3.4.3 Test Objectives

As described above, it is the goal of the system test to validate whether the complete system meets the specified functional and non-functional requirements (see chapter 3.7.1 and 3.7.2) and how well it does that. Failures from wrong, incomplete or inconsistent implementation of requirements should be detected. And requirements which are undocumented or have been forgotten should be identified.

3.4.4 Problems in System Test Practice

Excursion

In (too) many projects, the written documentation of the requirements is very incomplete or does not exist at all. Then, the testers face the problem that it is not clear what represents the system's correct behavior. This makes it hard to find defects.

Unclear system requirements

If there are no requirements, every system behavior is valid, and cannot be assessed. Of course, the user or the customer has a certain conception of what she expects of "her" software system. Thus, there *are* requirements. Yet, these requirements are not written down anywhere, they only exist in the heads of a few people that are involved in the project. The testers then have the not very desirable role of gathering information about the required behavior after the fact. A possible method to cope with such a situation is exploratory testing (see chapter 5.3 and for more detailed discussion [Black 02]).

Missed decisions

While the testers identify the original requirements, they will find out that different people have completely different views and ideas on the same subject. This is not surprising as the requirements have never been written down, reviewed and released during the project. As a consequence, the people responsible for the system test not only have to gather information on the requirements; they also have to enforce decision-making that should have been done many months before. This gathering of information is very cost and time consuming. The end of the tests and the release of the system will surely be delayed.

Projects may fail

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

3.5 Acceptance Test

All the test levels described so far represent testing activities that are run in the producer's responsibility. They are executed before the software is presented to the customer or the user for the first time.

But before installing and using the software in real life, another test level must be executed: the so-called acceptance test. Here, the focus is the customer's perspective and her judgment. This is especially important if the software was developed customer specific. The acceptance test might be the only test the customer is actually involved in or which she can understand. The customer may even be completely responsible for the acceptance test!

Acceptance tests can even be executed within lower test levels or distributed over several test levels:

A commercial off-the-shelf software product (COTS) may be acceptance tested when it is installed or integrated;
Acceptance testing of a component's usability may be done during component testing;
Acceptance testing of new functionality may come before system testing (using a prototype).

Typical forms of acceptance testing include the following:

1. Testing in order to determine if the contract is met
2. User acceptance testing

3. Operational (acceptance) testing
4. Field test (alpha and beta testing)

How much acceptance testing?

The extent of acceptance testing will vary considerably. It is dependent on application risk. If the software is developed customer specific, the risk is high and a full acceptance test as outlined above is necessary. The other extreme is an acquisition of a standard product that has been used for long time in a similar environment. Then, the acceptance test will consist of installing the system and maybe run a few representative →use cases. If the system shall cooperate with other systems in a new way, at least the interoperation should be tested.

3.5.1 Testing for Acceptance According to the Contract

If customer specific software was developed the customer (in cooperation with her vendor) will perform acceptance testing according to the contract. On the basis of the results of these acceptance tests she considers the ordered software system as being free of (major) deficiencies and whether the development contract or the service defined by the contract is accomplished. In the 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 formulated clearly and explicitly. Also any regulations which must be adhered to such as governmental, legal or safety regulations are to 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 which are relevant for acceptance showing the customer that the acceptance criteria of the contract are met.

As the suppher can have misunderstood the acceptance criteria, it is crucially important that the acceptance test cases are designed or at least thoroughly reviewed by the customer.

Acceptance test at the customer's site

In opposite to system test which takes place in the environment of the producer, acceptance test is run in customer's actual operational environment[18]. Due to the different testing environments, a test case that always worked correctly in system test can now suddenly fail. The final acceptance test also checks the delivery and installation procedures. The acceptance environment should be as similar as possible to

18 Sometimes acceptance test consists of two runs the first within system test environment, the second within customers environment

the later operational environment. But a test in the operational environment itself should be avoided, in order not to risk damaging running software systems.

For determining acceptance criteria and acceptance test cases the same methods as discussed before in the system test can be used. For administrative IT-systems especially business transactions with time constrained or periodic transactions (like a billing period) must be considered.

3.5.2 Testing for User Acceptance

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

Example: Different user groups

In the example of the VSR, the responsible customer is a car manufacturer. But the system will be used by the car manufacturer's dealers. The system's end users will be the employees of these dealers and their customers who want to purchase cars. In addition some clerks in the company's headquarter will work with the system, e.g. to put new price lists into the system.

Every user group should be included in the acceptance

The different user groups usually have completely different expectations to the new system. And if only one user group rejects the system because it finds it too awkward, this can lead to trouble with the introduction of the system. This may happen even if the system is completely OK from the technical or functional point of view. Thus, it is necessary to organize a user acceptance test for every user group. The customer usually organizes these tests, selecting test cases based on business processes and typical use scenarios.

Present prototypes to the users early

But if major user acceptance problems are detected during acceptance test, it is often too late to implement more than cosmetic measures. In order to prevent such disasters, it is advisable to let a number of representatives of the future users examine prototypes of the system already at an early stage of the project.

3.5.3 Operational (Acceptance) Testing

Operational (acceptance) testing assures the acceptance of the system by the system administrators. It may include the testing of backup-restore cycles, disaster recovery, user management, maintenance tasks, 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 test.

In such cases, after the system test the software producer may carry out a →field test. The objective of the field test is to identify influences from users environments that are not entirely known or that are not specified, and to eliminate them, if necessary.

Testing done by representative customers

Therefore, the producer delivers stable pre-release versions of the software to preselected customers that adequately represent the market for this software or whose operational environments appropriately cover possible environments.

These customers then either run test scenarios prescribed by the producer or they run the product on a trial basis under realistic conditions. They give feedback to the producer about the problems they encountered but also 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, beta tests at the customer's site.

A field test should of course not replace an internal system test run by the producer (even if some producers do exactly this). Only when the system test has proved that the software is stable enough, the new product should be given to potential customers for a field test.

3.6 Testing new Product Versions

Until now, it was assumed that a software development project is finished with the passing of the acceptance test and the deployment of the new product. Reality looks very different. 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. This chapter explains what must be considered when testing such new product versions.

3.6.1 Software Maintenance

Software does not wear out. Contrary to "classical" industry products, the purpose of software maintenance is not to maintain the ability to operate or to repair damages caused by heavy 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), or when defects are eliminated (corrective maintenance). Testing whether such changes work can be very difficult as the system's specifications often are out of date or missing. Especially in case of legacy systems.

Example: Analysis of VSR hotline requests

The VSR-System had been distributed and installed after intense testing. In order to find out weaknesses that had not been found until then, a 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 a platform with an old version of the operating system that is not recommended. In such environments, sometimes the host access causes system crashes.
2. Many customers consider the selection of extra equipment to be awkward, especially when they want to compare prices between different packages of extra equipment. Many users would therefore like to save equipment configurations and to be able to retrieve them after a change.
3. Some of the seldom-occurring insurance prices cannot be calculated at all, because implementing the corresponding calculation was forgotten in the insurance component.
4. Sometimes it takes more than 15 minutes before a car order is confirmed from the server. The system cuts the connection after 15 minutes in order to avoid having open unused connections. The customers are angry with this, because they waste a lot of time having to wait in vain for the confirmation of the purchase order. The dealer then has to repeat the order and mail the confirmation to the customer.

Problem 1 is the responsibility of the dealer, because he runs the system on a platform that the system was not intended for. Still, the software producer might change the program to make it able to run also on this platform, maybe in order to spare the dealer the cost of a hardware upgrade.

Problems like number 2 will always arise, no matter how well and complete the requirements were originally analyzed. This is due to the fact that the new system will generate many new experiences and therefore naturally, new requirements arise.

Improve the test plan

Problem 3 could have been detected during the system test. But testing cannot guarantee that a system is completely fault-free. It can only be a sample with a certain probability to reveal failures. A good test manager would check which kind of testing would have found this problem and will adequately improve or extend her 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 stopping the connection. Sometimes, it happens that there is a long waiting time because batch processes are run in the host computer. The fact that 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 even in the most mature software system:

1. The system is run under new operating conditions that were not predictable and were not planned.
2. The customers express new wishes.
3. Functions are necessary for seldom arising special cases that were forgotten.
4. Crashes, that happen rarely or only after very long uptime, are reported. These crashes 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 maintenance and software support. But the fact that maintenance is necessary in any case must not be used as pretext for cutting down on component, integration or system testing. Like "We must continuously publish updates anyway, so we don't need to take testing so seriously, even if we miss defects". Managers acting like that have not understood the true costs of failures.

Testing after maintenance

The overall test strategy is easy: anything new or changed should be tested. And to avoid side effects the remainder of the system should be regression tested (see chapter 3.7.4).

Testing after change of environment

Even if the system is not changed, but only its environment, maintenance testing is necessary.

E.g. if the system was migrated from one platform to another testing should repeat the operational tests within the new environment.

Testing for retirement

If a system shall be retired then also some testing is useful. Such testing for the retirement of a system should include the testing of data archiving or data migration into the future system.

3.6.2 Release Development

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

Example: Planning of the VSR development

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

1. New communication software is installed on the host in the car manufacturer's computing center. The VSR communication module must therefore be adapted to it.
2. Certain system extensions that could not be finished in release 1 are now delivered in release 2.
3 The installation base shall be extended to the European 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 neither come from defects nor from unforeseen user requests. So, they are not part of ordinary maintenance, but normal further product development.

The first point results from a planned change of a neighbor system. Point 2 is 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.

Therefore, a software product is not at all finished with the release of the first version. Instead, continuous further development is going on. An improved product version will be delivered at certain points of time, e.g. once a year. It is best to synchronize these →releases with the ongoing maintenance work. E.g. every half-year a new version is introduced: a maintenance update and a genuine functional update.

After each release, the project practically starts again running through all the project phases. Therefore, this is called iterative software development. Nowadays this is the usual way of developing software[19].

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. Like in maintenance testing anything new or changed should be tested, and the remainder of the system should be regression tested in order to find unexpected side effects (see chapter 3.7.4).

3.6.3 Testing in Incremental Development

Incremental development means that the project is done not in one (possibly large) piece, but as a series of smaller developments and

19 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])

deliveries. Functionality and requirements to its reliability will grow over time, from an early version only for the development group or for special users, to versions released to final customers later. Each increment, added to other increments developed previously, forms a growing partial system.

Incremental models try to reduce the risk of developing the wrong system, by delivering useful parts of the system early and getting customer feedback.

Examples for incremental models are: Prototyping, Rapid Application Development (RAD) [Martin 91], Rational Unified Process (RUP), Evolutionary Development [Gilb 05], use of 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], or SCRUM [Beedle 01].

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

The practical way to run such a project is to run several V-models after each other, where every next "V" reuses existing test material and adds the tests for everything new, or for higher reliability requirements.

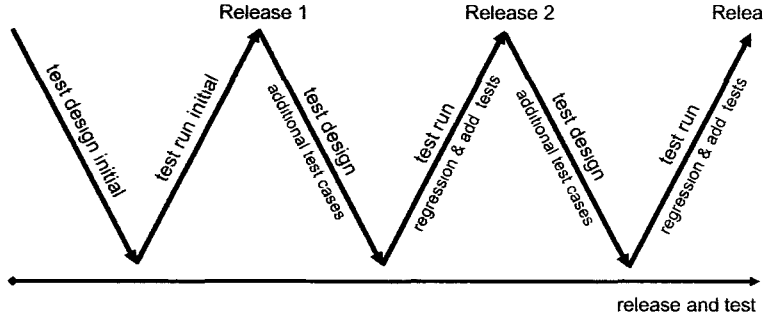


Fig. 3–5
Testing in incremental development

3.7 Generic Types of Testing

The former chapters gave a detailed view of testing in the software lifecycle, 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;
non-functional testing;
testing of software structure;
testing related to changes.

3.7.1 Functional Testing

Functional testing includes all kind of tests which verify a system's input-output behavior. To design functional test cases the black box testing methods from chapter 5.1 are used. And the test basis are the 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 the precondition for the system to be applicable at all. Characteristics of functionality, according to [ISO 9126], are suitability, accuracy, interoperability, security.

The (individual) customer's or the market's requirements to a software system are kept track of in a requirements management system (see chapter 7.1). Also text based requirements specifications are still in use. A format for this document is available in e.g. [IEEE 830].

The following text shows a part of the requirement paper concerning price calculation for the system VSR (see chapter 3.2.3 for the specification):

Example: Requirements to 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 calculated from current price lists and shown continuously.

Requirements-based testing

In →requirements-based testing, the released requirements are used 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 also reviewed. The testing of requirement 102 of the example shown above could look like this:

Example: Requirements-based testing

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

Usually, more than one test case is needed to test a functional requirement. Requirement 102 in the example contains several rules for different price calculations. These must be covered by a set of test cases (102.1 – 102.4 in above example). Using black box test methods (like 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 like shown above is mainly used in system test and acceptance testing. If a software system has the purpose to automate or support a certain business process of the customer, business-process-based testing or use-case-oriented testing are other similar suitable testing methods (see chapter 5.1.5).

Example: Testing based on business procedure

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

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

A business process analysis (which is usually elaborated as part of the requirements analysis) shows which business processes are relevant, and how often and in which context they appear. It also shows which persons, enterprises and external systems are involved. After that, 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 test 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 only interesting to see if they can choose and then buy a car. More important for the acceptance is often how easily she can use the system. This depends on how easy it is to work with the system, if it reacts fast enough, and if it returns clear output. Therefore, along with the functional criteria, the non-functional criteria must also be checked and tested.

3.7.2 Non-functional Testing

→Non-functional requirements describe not the functions, but 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 carry out its function. The implementation of such requirements has a great influence on customer and user satisfaction with the product and how much they like to use it. Characteristics of these requirements are, according to [ISO 9126]: reliability, usability, and efficiency. In an indirect manner, also the ability of the system to be changed and to be installed in new environments 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, as maintenance is a major cost driver!

According to [Myers 79], the following non-functional 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 of 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 it is overloaded.

Testing of security against unauthorized access, denial of service attacks etc.

Stability or reliability test 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, or wrong programming, or hardware failure etc., as well as examination of exception handling and recovery.
Testing of compatibility and data conversion: Examination of compatibility to given systems, import/export of data etc.
Testing of different configurations of the system, e.g. 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 output etc., always with respect to the needs of a specific group of users ([ISO 9241], [ISO 9126]).
Checking of the documentation for compliance with the system behavior (e.g. user manual and GUI).
Checking of 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 main problem in testing non-functional requirements is the often imprecise and incomplete character of these requirements. Expressions like "the system should be easy to operate" and "the system should be fast" are not testable in this form. Non-functional Requirements should be expressed in a testable way.

Hint

Representatives of the system test personnel should participate in early requirement reviews and make sure that every non-functional requirement can be measured and is testable.

Furthermore, there are many non-functional requirements that are so fundamental that nobody really thinks about mentioning them in the requirement paper (presumed matters of fact). Even such implicit requirements[20] must be validated because they may be relevant.

Example: Presumed requirements

The VSR-System is designed for the use on a market-leading operating system. It is obvious that the recommended or usual user interface conventions are followed for the "look & feel" of the VSR-GUI. The *DreamCar*-GUI (see

20. This is also true for functionality. The "of course the system has to do X" implicit requirement is a major problem for testing.

figure 3-3) violates these conventions in several aspects. Even if no particular requirement is specified, such deviations can and must be seen as faults or defects.

Testing non-functional requirements

In order to test non-functional characteristics, it makes sense to reuse existing functional 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 non-functional property must be observable in the corresponding test scenario. When executing the test scenario, the non-functional 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 non-functional system characteristics.

3.7.3 Testing of Software Structure

Structural techniques (white box testing) use information on the test object's internal code structure or architecture (statements or decisions, a calling hierarchy, menu structures). Also abstract models of the software may be used (e.g. a process flow model, or state transition model).

The objective is to design and run enough test cases to if possible completely cover all structure items. Structural techniques approaches are most used in component and integration testing. But can also be applied at system, system integration or acceptance testing (e.g. to cover menu structures). Structural techniques are covered in chapters 4.2 and 5 in more detail.

3.7.4 Testing related to Changes and Regression Testing

In case existing software is changed, defects are repaired or new parts are added, the changed parts must be →retested. In addition, there is the risk of side effects. In order to address them, existing test cases are repeated. These tests are called regression tests.

Regression testing

The regression test is a retest of a previously tested program following modification, to ensure that faults have not been introduced or uncovered as a result of the changes made.

Such faults often arise as unplanned side effects of program changes. This means regression testing may be performed at all test levels, and applies to functional, non-functional and →structural test-

ing. Test cases that are used in regression testing run many times and thus have to be well documented and reusable. They are therefore strong candidates for →test automation.

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

Volume of the regression test

1. Rerunning of all the tests that have detected faults which 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 integra ted (testing of new functionality).
4. Testing of the whole system (complete regression test).

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

Alterations can have unexpected side effects

If the test covers only altered or new code parts, the test neglects which 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 happen. This is particularly difficult for changes of systems with insuf ficient documentation or missing requirements. Unfortunately, this is often the case in old systems.

Complete regression test

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

In practice, a complete regression test is usually too time consuming and costly. Therefore, we are looking for criteria that can help to decide which old test cases can be omitted without losing too much information. As always, in testing this means balancing risk and cost. The best way to decide this is making an impact analysis for the changes, i.e. trying to determine where side effects may occur. The following strategies are often used to decide on this subject:

Selection of regression test cases

Only the high priority tests according to the test plan are repeated.

In the functional test, certain variations (special cases) are omitted.
Restrictions of the tests to certain configurations only (e.g. testing of the English product version only, testing of one operating system version only).
Restriction of 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], [Binder 99]. There, the authors not only describe special problems in regression testing object-oriented programs, but also elaborately describe the general principles of regression testing.

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 phase there is a corresponding test level;

The objectives of testing are changing, and 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 information as early as possible;

The number and intensity of the test levels may be tailored according to the specific needs of the project.

The V-model uses the fact that it is cheaper to repair defects short time after they have been introduced than after a long time. If defects remain undetected over several phases of the project, they lead to new defects in documents and products depending on the original defective product. This is, in general, more costly to repair the later a defect is detected. The defect leads to a so-called "ripple effect".
The component test tests single software components. The integration test tests the collaboration of these components. System tests examine the entire system from the perspective of the future users. In the acceptance test, the client checks the product for acceptance respective to the contract and acceptance by operation and the

users. If the system is supposed to be installed in many operational environments, then field tests provide an additional possibility to get experience with the system through running preliminary versions.

Defect correction (maintenance) and further development (enhancement) continuously alter and extend the software product throughout its life. All these altered versions must be tested again. Risk analysis must be used to determine the amount of the new tests as well as the regression tests.

There are several types of test: Functional testing, non-functional testing, testing of software structure, testing related to changes and regression testing.

4 Static Testing

Static examinations, like reviews and tool supported code and document analyses can be successfully used for quality improvement. This chapter presents the specific possibilities and techniques for that.

An often-underrated test method is the so-called static test, consisting of manual checking and static analysis. Contrary to the dynamic test (see chapter 5), the test object is not executed with test data but analyzed. This analysis can be done using one or several people to intensively inspect a document or using specific tools. All documents in a software development project can be inspected manually, even any document of value outside software projects. Tool-supported static analysis can only be done with documents that follow rules whose checking can be automated.

The main goal of all examination is to find defects and deviations from the existing specifications, defined standards or even the project plan. The results of these examinations are used additionally to optimize 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 course of the development, where they would otherwise result in expensive rework.

4.1 Structured Group Examinations

4.1.1 Foundations

Systematic use of the human capability to think and analyze

Reviews apply the human analytical capabilities to check and evaluate complex issues. This is done through intensive reading and trying to understand the documents that are examined.

There are different techniques for checking documents. They can be distinguished by the intensity, formality, the necessary resources (staff and time) as well as by their objectives. Below, the different techniques are explained in more detail. Unfortunately, there is no uniform terminology concerning static analysis techniques. The terms used here are analog to the terms in the ISTQB syllabus and [IEEE 1028] (see glossary in the appendix). Detailed descriptions can be found in [Freedman 90], [Gilb 96].

4.1.2 Reviews

Review is a common generic term for all the different human static analysis techniques as well as the term for a specific document examination technique. Another term, often used in the same meaning, is →inspection. However, "inspection" is 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, for example contracts, requirements definitions, design specifications, program code, test plans, manuals. Often, reviews are the only possibility to check the semantics of a document. Reviews use the colleagues of the author to give feedback. Because of this, they are also called →peer reviews.

Means to assure quality

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, in order to find mistakes and inconsistencies early. The verifying examinations (phase exit reviews) at the end of a phase in the general V-model normally use reviews. Eliminating defects leads to improved quality of the documents and has a positive influence on the whole development process, because development is continued with documents that have less or even no defects.

Positive effects

In addition to defect reduction, reviews additionally have the following positive effects:

Cheaper defect elimination. If defects are recognized and eliminated early, productivity in development is increased, because fewer resources are needed for defect recognition and elimination later, when it is substantially more expensive. These resources can be used for development instead (see chapter 3)

This result in shortened development time.

If defects are recognized and corrected early, costs and time needed for execution of 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 lifecycle of a product. For example, a review may detect and clarify inconsistent and imprecise customer wishes in the requirements. Foreseeable change requests after installation of the software system can thus be avoided.

A reduced failure rate during operation of the system 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 the products following later.
As several persons are involved in a review, a clear and understandable description of the facts is required. Often, already the necessity to formulate a document clearly lets the author find forgotten issues.
The whole team feels responsible for the quality of the examined object and the group will get a common understanding of it.

Potential problems

The following problems can arise: In a badly moderated review session, the author may feel that he himself and not the document is subject to critical scrutiny. Motivation to subject documents to a review will thus be destroyed. [Freedman 90] discusses extensively how to solve problems with reviews.

Reviews costs and savings

The costs caused by reviews are estimated to be 10–15 % of the development budget. The costs include the activities of the review process itself, the analysis of the review data and the effort for their implementation 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.

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 [Gilb96].

Hint

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 chapter 4.2)

The following factors are decisive for the success when using reviews (as suggested by [IEEE1028]):

Every review has a clear goal, formulated beforehand.
The "right" people are chosen as review participants, based on their subject knowledge and skills.

4.1.3 The General Process

The term review describes a whole group of static examinations. The different techniques are described in chapter 4.1.5. The process underlying all examinations is shortly described here in accordance with the IEEE Standard for Software Reviews [IEEE 1028].

A review requires six work steps: Planning, overview, preparation, review meeting, re-work, follow-up.

Planning

Reviews must certainly be planned

During overall planning, management must decide which documents in the software development process are 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 the work on it has been finished. In more formal reviews, entry criteria (and the corresponding exit criteria) may be set and checked.

Different viewpoints improve the result

Looking at documents from different perspectives may be more effective than an unfocused review. A review is in most cases more successful when the examined document is read from different viewpoints or if every person only checks special aspects. The viewpoints or aspects to be used should be determined while planning the review. It may also be decided not to look at the whole document, but to prioritize parts with the highest risk or to review samples only, in order to check the general quality of the document.

If an overview meeting is considered necessary, time and place must be chosen.

Overview

The overview (or kickoff) serves to provide the people involved in the review with all necessary information. This can happen through a written invitation or a first meeting when the review team is organized. The purpose is to inform about the document to be reviewed ("the review object") and the significance and the objective of the planned review. If the involved people are not familiar with the domain or application area of the review object, then there can be a short introduction to the material, as well a description of how it fits into the application or environment.

In addition to the review object, the people involved must have access to other documents. These include the documents that must be used 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 baseline. Furthermore, review criteria (for example checklists) are very useful in order to support a structured process.

Preparation

Intensively study of the review object

The members of the review team must prepare individually for the review meeting. A successful review meeting is only possible with adequate preparation. The reviewers intensively study the review object and check it against the documents given as a basis for it. They note deficiencies, questions, or comments.

Review meeting[21]

The review meeting is led by a review leader or →moderator. Managing and participating in reviews requires good people skills in order to protect the participating people and motivate them to best contribute to the review.

The review leader must ensure that all experts will be able to express their opinion without fear, that the product will be evaluated and not the author, and that conflicts will be prevented or resolved

Usually, the review meeting is limited. The objective is to decide if the review object has met the requirements and complies with the standards, as well as to find defects. The result is a recommendation to accept, repair or rewrite the document. All the reviewers should agree upon the findings of this evaluation and the general result.

Here are the following general rules for a review meeting[22]:

1. The review meeting is limited to two hours. If necessary, another meeting is called, not before the next day.
2. The moderator has the right to cancel or discontinue a meeting if one or more experts (reviewers) don't appear or if they are insufficiently prepared.
3. The document subjected to review, the examination object, is sub ject 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. Justification or explanation of her decisions is however partially seen as legitimate and helpful.)
4. The moderator should not be a reviewer at the same time.

21 IEEE Standard 1028 calls this "Examination"

22 Some of these rules do not apply to all kinds of reviews of the IEEE Standard 1028

5. General style questions (outside the guidelines) shall not be discussed.
6. Developing solutions and their discussion is not a task of the review team.
7. Every reviewer must have the opportunity to adequately present 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 must be weighted[23] as:
    Critical defect (the review object is not suitable for its purpose, the defect must be corrected before the object is approved),
    Major defect (the usability of the review object is affected, the defect should be corrected before the approval)
    Minor defect (small deviation, hardly affects the usage)
    Good (flawless, this area should not be changed during rework)
11. The review team shall make a recommendation for 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. At the end, all the session participants should sign the protocol

Protocol and summary of the results

The protocol contains a list of the issues / findings that were discussed in 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 chapter 4.1.4), a short summary of the most important issues and the result of the review with the recommendation of the reviewers. When executing a formal review, formal exit criteria may be checked.

Rework

The manager decides whether she follows the recommendation or if she takes a different decision, for which she would have to take the whole responsibility. Usually, the author will eliminate the defects on the basis of the review results.

---

23. See chapter 6.6.3: →Severity class 2 and 3 defects can be seen as major defects and class 4 and 5 as minor defects.

Follow up

Correcting the findings must be followed up, usually by the manager or moderator or by someone especially assigned this responsibility.

Second review

If the result of the first review was not acceptable, another review should be scheduled, the process described here can be rerun, but normally, it is done in a shortened way, only checking changed areas.

A thorough evaluation of the review meetings and their results should then be done to improve the review process and to adapt the used guidelines and checklists to the specific conditions and to keep them up to date. In order to achieve this it is necessary to collect and evaluate measurement data.

Deficiencies in the software development process

Recurring or frequently occurring defect types point to deficiencies in the software development process or in the technical knowledge of the particular people. Necessary improvements of the development process should be planned and implemented. Such defect types should be included in the checklists. Lack of technical knowledge must be compensated by training.

4.1.4 Roles and Responsibilities

The description of the general approach already gave some information on the roles and responsibilities. This section presents the involved people.

Manager

The development manager selects the objects to be reviewed and makes sure that the base documents as well as the necessary resources are available. She chooses the participating people.

Still, representatives of the management level should not participate in the review meeting in case the author or some reviewers are scared of the possibility that the manager may use the review to evaluate them as a person. Thus, a "free" discussion among the review participants is probably made impossible. Another reason is that the manager normally does not have the necessary understanding of technical documents. In a review, the technical content is to be checked. Thus, the manager is not qualified to participate. (Management) reviews of project plans and the like are a different thing.

Moderator

The moderator is responsible for the administrative tasks pertaining to the review, for planning and preparation, shall ensure that the review is conducted in an orderly manner and meets its objectives, is responsible for collecting review data, and shall issue the review report.

The moderator is crucial for the success of the review. First and foremost, she must be a good meeting leader, leading the meeting efficiently and in a diplomatic way. She must be able to stop unnecessary discussions without hurting the participants, to mediate when there are contrary points of view, and be able to understand discussions "between the lines". She must be neutral and must not state her own opinion about the review object.

Author

The author is the creator of the document that is subjected to a review. If several people have been involved in the creation, one main responsible should be appointed; she takes over the role of the author.

The author is responsible for the review object meeting its review entry criteria (generally that the document is in a reasonably complete state), for contributing to the review based on her special knowledge and understanding of the document, and for performing any rework required to make the review object meets its review exit criteria

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 only done to help improve the product.

Reviewer

The reviewers, sometimes also called inspectors, are several (normally maximum five) technical experts that shall check the review object after individual preparation.

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

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

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

Recorder

The recorder shall document the findings (problems, action items, decisions, and recommendations) made by the review team. The recorder must be able to record in a short and precise way. She must write down the essence of the discussion. This may not be easy as contributions are often not clear or well expressed in the way they were originally made. It can make sense to have the author write the protocol. She knows exactly how precisely and how detailed the contribu-

tions of the reviewers need to be recorded in order to have enough information for follow up.

Possible difficulties

Reviews may fail be due to several causes:

Reasons for reviews to fail

The required persons are not available or do not have the required qualification or technical aptitude. This may be solved by training or by using qualified staff from consulting companies. This is especially true for the moderator, because she must have more psychological than technical skills.

Wrong estimates during resource planning by management may result in time pressure, which then cause 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 the reviewer does not realize the importance of the review and its great effect for quality improvement and the review fails because of this, then the benefit of the review has to be shown by figures that prove the efficiency.

A review can also fail because of missing or insufficient documentation. Before the review it must be checked if all the needed documents exist and if they are descriptive enough. Only if this is the case, a review can be carried out.

The review process cannot be successful if management support is lacking, 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 hints for solving these problems are described in [Freedman 90].

4.1.5 Types of Reviews

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

Reviews pertaining to technical products or partial products that have been created during the development process and

Reviews that analyze project plans and the development process.

Excursion

The purpose of a →management review, [IEEE 1028][24] (or project review) is to monitor progress, determine the status of plans and schedules, confirm requirements and their system allocation, or evaluate the effectiveness of management approaches used to achieve fitness for purpose

The project as a whole as well as the determination of its current state is the review object The state of the project is evaluated with respect to technical, economical, 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 "post-mortem"-analysis, in order 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 particular descriptions, the focus is laid on the main differences to the basic review process (see chapter 4.1.3).

Walkthrough

A walkthrough[25] is an informal review method with the purpose of finding defects, ambiguities and problems in the written documentation. The author presents the document to the reviewers in the review meeting.

[IEEE 1028] additionally mentions the purpose of educating an audience regarding a software product. Main objectives are to find anomalies, to improve the product, to consider alternative implementations and to evaluate conformance to standards and specifications.

The focus of the walkthrough is the meeting (without time limit). The preparation has the smallest amount compared to the other types of reviews; it can even be omitted sometimes[26].

Discussion of typical usage situations

In the meeting, the author presents the product. Usually, typical use cases, also called scenarios, are walked through according to the course of events. Also, single use cases can be simulated. The reviewers try to reveal possible defects and problems by spontaneously asking questions.

Suitable for small development teams

This process is suitable for small development teams of 5 to 10 persons and causes little effort, because preparation and follow up

24 In [ISO 8402] 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 quality policy and objectives"

25 Also called "structured walkthrough".

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

do not take many resources and are not mandatory. A walkthrough can be used for checking "non-critical" documents.

Due to the fact that the author chairs the meeting, she has a great influence. This can have a detrimental influence on the result, if the author does not want a discussion of the critical parts of the review object. The author is responsible for follow up; there is no more checking involved.

The following different approaches are also possible for a walkthrough: The reviewers prepare before the meeting, the results are written in a protocol, and the findings are listed instead of letting the author note them. In practice there is a wide variation from informal to formal walkthroughs.

Inspection

Formal process

The inspection is the most formal review. It follows a formal prescribed process. Every person, normally chosen from the direct colleagues of the author, has a certain defined role. The course of events is defined by rules. Checklists containing inspection criteria (formal entry- and exit criteria) for the individual aspects are used.

The focus is finding unclear points and possible defects, measuring document quality, and improving the quality of the product and the development process. The objectives of the inspection are determined during planning, and only a specific number of aspects will be focused. The inspection object is checked with respect to formal entry criteria before start. The inspectors prepare themselves using procedures, standards and checklists.

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

Inspection meeting

The inspection meeting follows this agenda: A moderator leads the meeting. The moderator first presents the participants and their roles, as well with a short introduction to the topics to be checked. The moderator asks every participant if he or she is prepared well enough. It may be asked how much time the reviewer has used and how many issues were found. The group may review the checklists chosen for the inspection in order to secure that everyone is well prepared for the meeting.

Issues of a general nature are then discussed and written to the protocol.

A reviewer[27] presents the contents of the inspection object in a short and logical way. If it is considered useful, passages can also be read aloud. The reviewers ask questions during this procedure, and the selected aspects of the inspection are intensely checked. The author answers questions, but remains passive in general. If author and reviewer disagree about an issue, this may be discussed at the end of the meeting.

The moderator must intervene if the discussion is going out of control. The moderator also makes sure that the meeting covers all aspects to be evaluated as well as the whole document. The moderator makes sure that the recorder keeps track of all the issues and ambiguities that are detected.

At the end of the meeting, all recorded items are reviewed for completeness. Issues where there was disagreement are discussed in order to resolve if they are defects or not. If no resolution is reached, this is written in the protocol.

Finally, a judgment is reached about the inspection object as a whole. It is decided if the inspection object must be reworked or not. In inspections, follow-up and re-inspection are formally regulated.

Additional assessment of the development and inspection process

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

Technical review

Does the review object fulfill its purpose?

In a technical review, the focus of attention is compliance of the document with the specification, fitness for its intended purpose and compliance to standards. During preparation, the reviewers inspect the review object according to the specified review criteria.

Technical experts as reviewers

The reviewers must be technically qualified experts. Some of them should not be project participants, in order to avoid "project blindness". Management does not participate. Background for the review is only the "official" specification and the specified tasks for the review. The reviewers write down their comments and pass them to the moderator before the review meeting[28]. The (in the ideal case trained) moderator sets the priority for these findings according to their presumable

27. IEEE 1028 says "reader".
28. In [IEEE 1028] this also applies to inspection.

importance. During the review meeting, only selected important remarks are discussed.

High preparation effort

Most effort lies in the preparation work. During the meeting, normally not attended by the author, the recorder notes all the issues and prepares the final documentation of the results.

The review result must be approved unanimously by all involved 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 a management responsibility. If the review is highly formalized, entry and exit criteria of the review 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.

Informal review

The informal review is a light version of a review. However, it more or less follows the general procedure for reviews (see chapter 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 of time. Often, there is no meeting or exchange of the findings. In such cases, the review is just an author-reader cycle. The informal review is a kind of cross-read of one or more colleagues. The result need not be explicitly documented; a list of remarks or the revised document is enough.

. Pair programming, buddy testing, code swapping and the like are all kinds of informal reviews. The informal review has a high acceptance due to the little effort it takes and is very common.

Selection criteria

Selection of type of review

Which type of review should be used when depends very much on the requested quality and the effort that has to be spent. It depends on the project environment and specific recommendations cannot be given. It must be decided in the particular case which type of review is appropriate. Below, some questions and criteria are given that should help to make the selection of specific review types easier:

The form in which the result of the review should be presented can help select the review type. Is a detailed documentation necessary or is it enough to implement the checking results informally?

Will it be hard or easy to find a date and time for the review? It can be very hard to bring together 5 or 7 technical experts for one or more meeting.
Is it necessary to have technical knowledge from different disciplines?
How qualified review participants are necessary? Will the reviewers be motivated?
Is the preparation effort appropriate with respect to the benefit of it (the expected result)?
How formally written is the review object? Is it possible to run tool-supported analyses?
How much management support is available? Will management curtail reviews when the work is done under time pressure?

Notes

As we already 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.

Types of reviews depend on the organization where they are used

It can be said that generally, the type of a review is very much determined by the organization that uses it. The reviews are tailored for the specific needs and requirements of a project. This has a positive influence on their effectiveness.

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 it is suggested in →Extreme Programming can be regarded as a permanent "two-person-review" [Beck 00].

In distributed project teams, there may be difficulties organizing review meetings. Modern ways of organizing reviews include structured discussion by Internet, video and telephone conferences etc.

Success Factors

When using reviews the following factors are decisive for the success and have to take into consideration:

Reviews serve the purpose of improving the examined documents. Detecting findings like unclear points and deviations is a wanted and required effect. The findings must be formulated in a neutral and objective way.

Human and psychological factors have a high influence in a review. It should be achieved that the author of the examined document feels the review as a positive experience.
Depending on the kind and level of the examined document and the state of knowledge of the participating people a different but appropriate kind of review should be chosen.
Checklists and guidelines are used in order to increase the effectiveness of detecting findings during reviews.
Training is necessary. This is especially true for more formal kinds of reviews, like inspections.
Management can support a good review process by planning sufficient resources (time and personnel) for document reviews in the software development process.
A very important aspect for the successful use of reviews is continuous learning from the reviews themselves, i.e. review process improvement.

4.2 Static Analysis

Analysis without execution of program

The objective of static analysis is, like in reviews, to reveal defects or parts in a document that are defect-prone. However, tools do the analysis. For example, even spell checkers can be regarded as a form of →static analyzers that find mistakes in the texts and therefore contribute to quality improvement. The term "static analysis" points to the fact that this form of checking does not contain 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.

Formal documents

The document to be analyzed must follow a certain formal structure in order to be checked by a tool. Static analysis only makes sense with the support of tools. Formal documents can be for example the technical requirements, the software architecture or the software design, as, for example, the modeling of class diagrams in UML[29]. Generated outputs in HTML[30] or XML[31] can also be subjected to tool supported static analysis. Formal models developed during the design phases can also be analyzed and inconsistencies be detected. Unfortunately, in practice, the program code is often the one and only formal document of the software development that can be subjected to static analysis.

29. UML – Unified Modeling Language [URL: UML]
30. HTML – HyperText Markup Language [URL: HTML]
31. XML – Extensible Markup Language [URL: XML]

Static analysis tools are typically used by developers in order to check if guidelines or programming conventions are adhered to, before or during component or integration testing. During integration testing, adherence to interface guidelines is analyzed.

Analysis tools often produce a long list of warnings and comments. In order to effectively use the tools, the mass of generated information must be handled intelligently for example by configuring the tool. Otherwise it may happen that the tools will not be used.

Static analysis and reviews

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

Hint

If documents are formal enough to allow tool-supported static analysis, then this should definitely be performed before the document reviews because faults and inconsistencies 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 located and removed discrepancy increases the quality of the document.

Not all defects can be found using static testing, though. Some defects show their effect only when the program is executed, that means at run time 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 run time. In static analysis, this defect cannot easily be found, except for when this variable is assigned the value zero by a constant having this value. Alternatively, all possible paths through the operations are 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 only possible with static analysis (or reviews).

The compiler is an analysis tool

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

The following defects and constructions that bear the danger of producing problems can be detected by static analysis:

Syntax violation
Deviation from conventions and standards
→Control flow anomalies
→Data flow anomalies.

Finding security problems

Static analysis can be used in order to detect security problems. Many security holes occur because certain error-prone program constructs are used and necessary checks are not done. Examples are lack of buffer overflow protection or failing to check that input data may be out of bounds. Tools can find such deficiencies, because they often have a certain "pattern" which can be searched for and found by tools.

4.2.1 The Compiler as 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.

Examples are:

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
Detecting over- or underflow of field boundaries (static addressing)
Checking of 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. Further investigation is necessary.

4.2.2 Examination of Compliance to Conventions and Standards

The compliance to conventions and standards can also be checked with tools, for example if most programming regulations and standards have been respected. This way of inspecting takes little time and almost no personnel resources. In any way, only guidelines that can be verified by tools should be accepted in a project. Every other regulation normally proves to be bureaucratic waste anyway. Furthermore,

there is often another advantage: If the programmers know that the program code is checked for compliance to the programming guidelines, their readiness to work according to the guidelines is much higher than without an automatic test.

4.2.3 Data Flow Analysis

Data use analysis

→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 detect defects. Instead, we speak of →anomalies or data flow anomalies. Anomaly means an inconsistency that can lead to failure, but does not necessarily do so. An anomaly may be flagged as a risk.

Data flow anomalies are for example reading variables without previous initialization or not using the value of a variable at all. The usage of every single variable is inspected during the analysis. 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 for anomalies

The different anomalies are explained referring to the following example (in C++). 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 that 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 analysis of the usage of the single variables, 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 use of the first value (before the second assignment) has been forgotten.
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 only valid inside the function.

Data flow anomalies are usually not that obvious

In this example, the anomalies are obvious. But it has to 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, e.g. a review. A tool for analyzing data flow can, however, detect the anomalies.

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 Control Flow Analysis

Control flow graph

In figure 4-1, a piece of program 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.

Changes in the course of program execution are represented by decisions, e.g. 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 sequence in a piece of program can easily be understood and possible anomalies can be detected. These anomalies could e.g. be jumps out of a loop body or a program piece that has several exits. These anomalies need 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 in 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, a revision of the program text should be done, because complex sequence structures often bear a great risk of errors.

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 there is a statement that does not have a predecessor, then this statement is unreachable (so-called dead code). Thus a defect or at least an anomaly is detected. Only for the first and the last statements of a program, there may not be a 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 mentioned analyses, static analysis tools also 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 the 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, a closer look at a certain metric will be taken: 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 the cyclomatic number can be computed like this:

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

v(G) = cyclomatic number of the graph G
e = number of edges in the control flow graph
n = number of nodes in the control flow graph

Example of the calculation of the cyclomatic number

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

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

with
e = number of edges in the graph = 17
n = number of nodes in the graph= 13

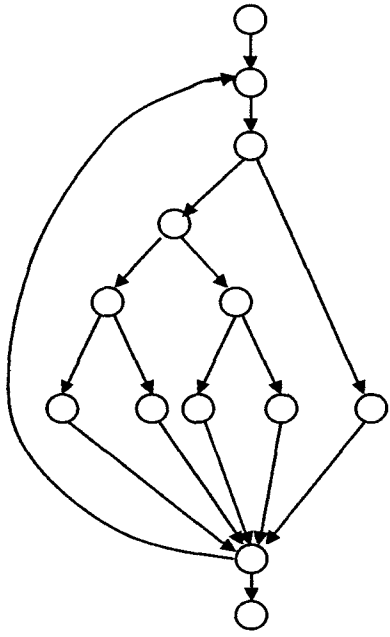


Fig. 4–1
Control flow graph for the calculation of the cyclomatic number (identical to figure 2-2)

The value of 6 is according to McCabe acceptable and in the middle of the range. We assume that a value higher than 10 can not be tolerated and rework of the program code has to take place.

The cyclomatic number gives information about the testing effort

The cyclomatic number can be used to estimate the testability and the maintainability of the particular program part. The cyclomatic number specifies the number of independent paths in the program part[32]. If 100 % branch coverage (see chapter 5.2.2.) is intended, then all these independent paths of the control flow graph have to be exe-

32. All linearly independent paths are meant.

cuted 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 a certain program part.

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

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. The results are recorded.

A fundamental review process consists of the following activities: Planning, overview, preparation, review meeting, re-work and follow-up. The roles of the participants are manager, moderator, author, reviewer and recorder.

There are several types of review. 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 only little preparation for the meeting. The walkthrough is suitable especially for small development teams, for discussing alternatives, and for educating people.

The inspection is the most formal review process. Preparation is done using checklists, there are defined entry and exit criteria, the meeting is chaired by a trained moderator, and data are collected and used for quality improvement of both development and the inspection process itself.

In the technical review, the individual reviewers' results are presented to the review leader before the meeting. The meeting is then prioritized by assumed importance of the individual issues. The author does not participate. Checking is done using documents only.

The informal review is not based on a formal procedure. It is not prescribed in which form the results have to be presented. Because

this type of review can be performed without much effort, its acceptance is very high and in practice, it is very widely spread.
Generally, the type of reviews used is very much determined by the specific environment, i.e. the specific organization and project that the review is used in. The reviews are tailored to meet the specific needs and requirements, which increases their efficiency. It is important to establish a co-operative collaborative atmosphere between the people involved in the software development.
In addition to the reviews, a whole series 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 this.
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 can also show violation of standards and other conventions.
Tools are available for detecting anomalies in the data and control flow of the program. Useful information about control and data flow is generated. This information often points to parts that could contain defects.
Metrics are used to measure quality. The cyclomatic number 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 reviewing. Static analyses provide cheap means to detect defects and the reviews get cheaper.

5 Dynamic Analysis – Test Design Techniques

This chapter describes techniques for testing of software by executing the test-objects on a computer. It presents the different techniques for specifying test cases and for defining test exit criteria and then explains them by examples. These techniques are divided into black box testing and white box testing. Additional test design methods conclude this chapter.

Execution of the test object on a computer

In most cases, 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) must be executable. It is provided with input data before it is executed. In the lower test stages (component- and integration testing) the test object cannot be run alone, but must be embedded into a test bed to obtain an executable program (see figure 5-1).

Test bed necessary

The test object will usually call different parts of the program through predefined interfaces. These parts of the program are substituted by placeholders, called stubs whenever these parts are not implemented yet and therefore not ready to be used or if they are supposed

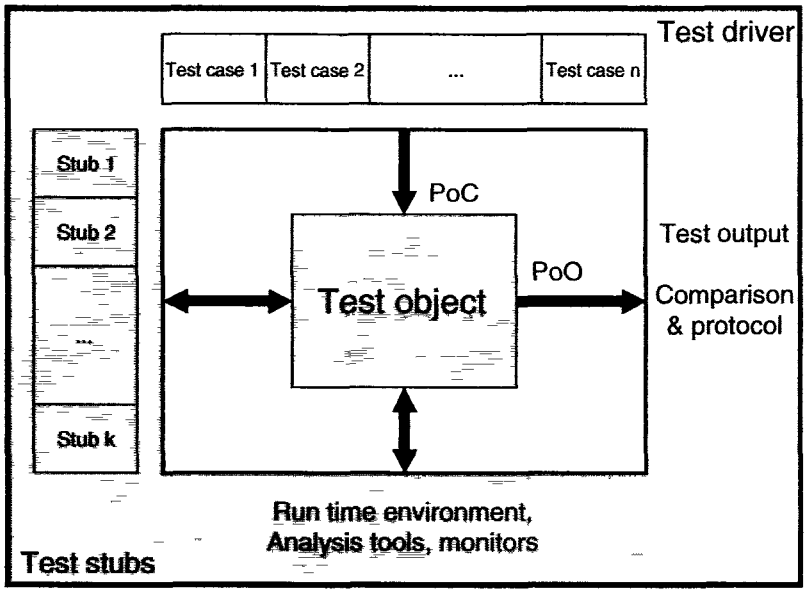


Fig. 5-1
Test bed

to 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[33].

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. The test driver does this. Driver and stub combined establish the test bed, which constitutes the executable program together with the test object.

The tester must often create the test bed himself 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 (chapter 7.1.4). Having such an executable test object empowers the tester to start the dynamic analysis.

Systematic approach at determination of the test cases

When executing a program, testing must expose failures and verify as many requirements as possible with as little expense as possible. To reach this goal it is necessary to choose a systematic approach. Unstructured testing, usually "by gut feeling", does not offer any guarantee. The tester should test as many situations as possible, but better all situations processed differently by the test object.

Incremental 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 (normally chaining together several test cases)

This work can be done in a very informal way, for example undocumented, or in a formal way as described in this chapter. The degree of formality depends on several factors such as: Application area of the system (for example safety critical software), maturity of the development and test process, time constraints and knowledge of the project

It must be determined how the individual requirements and the test cases relate to each other. Thus it is possible to determine the coverage of the requirements by the tests. It will also be easier to estimate the effect of requirement changes on the test (implementing new test cases or changing existing ones).

→Test case specification

Part of the specification of the individual test cases is determining test input data for the test object. They are determined by the methods described in this chapter. However, the preconditions for executing the test case are important, too, and the expected results and expected post conditions, in order to decide if there is a failure (detailed descriptions can be found in [IEEE 829]).

Determine expected result and behavior

The expected results (output, change of internal states, etc.) should be determined before executing the test cases. Otherwise it often happens that an incorrect result is interpreted as correct, thus missing to detect a failure.

Test case execution

It does not make much sense to execute one and one test case. Test cases should be grouped in such a way that a whole sequence of test cases is executed (test sequence or test scenario). Such a test sequence is documented in a →test procedure or test instruction. The document normally groups the test cases by topic or by test objectives. It should also be possible to find information about the test priorities and technical and logical dependencies between the tests and regression test cases. Finally, the timing of the test execution (assigning tests to testers and determining the point of time for execution) is described in the →test schedule.

In order to be able to execute a test sequence we need a →test script. The test script contains, most often in a programming language or a similar notation, instructions for automatically executing the test sequence. The corresponding preconditions can be set and the actual and expected results can be compared in the test script. JUnit is an example of a framework, which allows to easily program test scripts in Java [URL: xunit].

Black box and white box techniques

There exist several different approaches for testing the test object. They can be categorized in two groups: black box and white box[34] testing. To be more precise: Test case design techniques, because these techniques support the identification of the respective test cases.

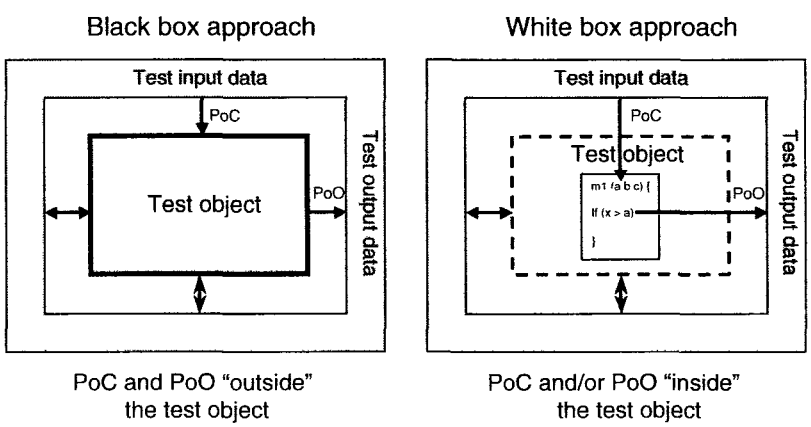
Using black box testing, the test object is seen as a black box. Test cases are derived from the specification of the test object. The behavior of the test object is watched from the outside (PoO – →Point of Obser-

34 Sometimes called "glass box testing" or "open box testing", because of the lack of transparency in a white box. Nevertheless, these terms are not in widespread use

vation is outside the test object). It is not possible to control the operating sequence of the test object besides choosing the adequate input test data (the PoC – →Point of Control is situated outside of test object, too). Test cases are designed only by using the specification or the requirements of the test object.

In white box testing the source code is known and used for test design. While executing the test cases, the internal processing of the test object as well as the output is analyzed (the Point of Observation is inside of the test object). Direct intervention in the process of the test object is possible, but should be used in special situations, e.g. to execute negative testing when the component's interface is not capable of initiating the provoked failure (the Point of Control can be located inside the test object). Test cases are designed to cover the program structure of the test object (figure 5-2).

Fig. 5-2
PoC and PoO at black box and white box techniques



White box testing is also called structural testing, because the test designer considers the structure (component hierarchy, flow control, data flow) of the test object. The black box testing techniques are known as functional or behavioral testing techniques, because of the observation of the input/output behavior [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.

Black box testing is predominantly used for higher levels of testing even though it is reasonable in component tests. Any test design before the code is written (test-first programming, test-driven development) is essentially black box driven.

Most test methods can clearly be assigned the two categories. Some have elements of both. These are sometimes called "grey box techniques". In the following two chapters, black box and white box techniques are described in detail. Intuitive or experience based testing can be said to be a black box technique. However it is described in a different chapter, as it is not a systematic technique.

5.1 Black Box Testing Techniques

Using 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. A test with all possible input data combinations would be a complete test. But this is unrealistic considering the enormous number of combinations (chapter 2.1.4). Test design must make a reasonable selection of all possible test cases. There are a couple of methods to do exactly that. They will be introduced below.

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 →equivalence classes (equivalence class partitioning). An equivalence class is a group of data values where the tester assumes that the test object processes them in the same way. The test of one representative of the equivalence class is seen as sufficient because it is assumed that for any other input value of the same equivalence class the test object will not show a different reaction. 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 Christmas bonus from chapter 2.2.2 is revisited here to clarify the facts. As a reminder: The program shall calculate the Christmas bonus of the employees depending on the affiliation to the company. The following text is part of the description of the requirements: "Employees receive 50 % of their monthly income as a Christmas bonus if they have been working for the company for more than 3 years. Employees who have been employed for more than 5 years receive 75 %. With more than 8 years of employment the employee is awarded with 100 % as bonus."

Four different equivalence classes with correct input values (correct, or "valid" equivalence classes, VEC) can be derived very easily from the calculation of the bonus by considering the length of employment.

Tab. 5-1 Correct equivalence classes and representatives

Parameter	Equivalence classes	Representative values
Bonus calculation program, duration of employment in years	vEC_1: 0 <= x <= 3	2
	vEC_2: 3 < x <= 5	4
	vEC_3: 5 < x <= 8	7
	vEC_4: x > 8	12

In chapter 2.2.2, the input values 2, 4, 7, 12 (compare 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 1, 6, 9, and 17 does not lead to further insights and therefore not exposes further failures. With this assumption, it is not necessary to execute those extra test cases.

Equivalence classes with incorrect values

Besides the correct input values, incorrect 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 before, there are the following two invalid equivalence classes (iEC)[35]:

Tab. 5-2 Invalid equivalence classes and representatives

Parameter	Equivalence classes	Representative values
Bonus calculation program, duration of employment in years	iEC_1: x < 0 ("Negative" – thus incorrect – staff membership in a company)	-3
	iEC_2: x > 70 (Unrealistically long and incorrect staff membership in a company[a])	80

a. The value 70 is chosen rather randomly by judging nobody would be employed for such a long time. The maximum value for company affiliation has to be aligned with the customer's opinion.

Systematic derivation of the test cases

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 inputs should be determined. This domain is then partitioned into equivalence classes. First, the subdomain of correct inputs is found. This is the equivalence class of all correct input values. The test object should process these values according to the specification. The values outside of this domain are seen as equiva-

35. A correct term would be "equivalence classes for invalid values" instead of "invalid equivalence class", because the equivalence class in itself is not invalid, only the values of this class, for the specified input.

lence classes with incorrect input values. For these values as well, it must be tested how the test object behaves.

Further partitioning of the equivalence classes

The next step is to refine the equivalence classes. If the test object's specification tells 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 all different requirements correspond to an equivalence class. For every single equivalence class, a representative value should be chosen for testing.

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 also be used for the output data. Admittedly, identification of the individual test cases is more expensive because for every output-representative (output value) the corresponding input value combination causing this output has to be determined. For the output values as well, the equivalence classes with incorrect values must not be disregarded.

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

Boundaries of the equivalence classes

Promising 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 "... more than 3 years..." within the requirements might mean the value 3 being inside (EC: x >= 3) or outside of the equivalence class (EC: x > 3). An additional test case with x = 3 might detect a misinterpretation and therefore failure. Chapter 5.1.2 discusses the analysis of the boundary values for equivalence classes in detail.

Example: Equivalence class construction for integer values

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

Tab. 5-3
Equivalence classes for integer input values

Parameter	Equivalence classes
extras	vEC_1: [MIN_INT,.., MAX_INT] [a] iEC_1: NaN (Not a Number)

a. MIN_INT and MAX_INT each describe the minimum and maximum whole number that the computer is able to use. These can vary depending on the used hardware.

Notice that the domain contrary to plain mathematics is limited on a computer by its maximum and minimum. Using values outside the computer domain often lead to failures, because this situation is not handled 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 non-numeric value[36]. 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). Part of this equivalence class are the floating point numbers because it is expected that the program reacts with an error message to inputs such as "3.5". In this case, the equivalence class partitioning method does not require any further subdivision, as the same reaction is expected in every case of wrong input. However, an experienced tester will always include a test case with a floating-point number in order to determine if the program maybe rounds the number and continues with the rounded integer number. The basis of such an additional test case is intuition or experience based testing (see chapter 5.3).

It is reasonable to divide the equivalence classes with correct values further because negative and positive values often must be treated differently. Zero is a further input value that often leads to failures. Therefore, it is often interesting to test.

Tab. 5–4 Equivalence classes and representatives for integer values

Parameter	Equivalence classes	Representatives
extras	vEC_1: [MIN_INT, ..., 0[[a]	-123
	vEC_2: [0, ..., MAX_INT]	654
	iEC_1: NaN (Not a Number)	"f"

a. '[' 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 values for the three equivalence classes have been randomly chosen. The boundary values (see chapter 5.1.2) of the respective equivalence classes should be added: `MIN_INT`, `-1`, `0`, `MAX_INT`. The equivalence class for incorrect values has no boundary values.

Result: For the test of the integer parameter extras, the equivalence class method taking in account boundary values generates the following seven test values:

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

36. It depends on the given programming language and the used compiler which incorrect values the compiler or run time system is able to identify, e.g. while attempting to call the function from the test driver. In the example, it is assumed that the compiler or run time system cannot recognize incorrect values and that the processing of those values has to be examined in the dynamic test.

For every input value the expected outputs or reactions of the test object should be defined in order to decide after test execution whether a failure occurred or not.

Equivalence classes of input values, which are not 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 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 traveler can be a child, a teenager, an adult, a student, a person on welfare or a retired person. If the test object has to react differently to each kind of traveler, then every possibility has to be verified with an additional test case. If there is no requirement for different reactions for each person type, then one test case might be sufficient. In this case, any randomly chosen but correct value can be chosen for the traveler.

If the test object is the component that calculates the fare, and the fare depends on the type of person, then certainly six different test cases for the traveler must be provided. It is probable that the fare is calculated differently for each traveler. Details must be looked up in the requirements. Each calculation must be verified by a test to prove the correctness of the calculations and to find failures.

For the test of the component that handles the seat reservation, it might be sufficient to choose only one representative, e.g. an adult, for the traveler. Presumably, it is not relevant if a teenager or a retired person takes the seat. Here, as well, the requirements may tell otherwise and should be analyzed. The tester should be aware, though that in case she executes the test with the input "adult" only she will not be able to say anything about the correctness of the seat reservation for any of the five other traveler types.

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 in 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 possibly must 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 conditions 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

Combination of the representatives

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.

In order to specify a test case, each parameter must be assigned an input value. For this purpose, a decision must be made about which of the available values should be combined together to form test cases. In order to guarantee triggering all test object reactions (modeled by the equivalence class division), the input values, i.e. the representatives of the according equivalence classes, must be combined using the following rules:

Rules for test case determination

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 "positive test case".

Separate test of the invalid value

The representative value of an invalid equivalence class can only be combined with (arbitrary) representatives of other *valid* equivalence classes. Thus, every invalid equivalence class leads to an additional "invalid test case" or "negative test case".

Restriction of the number of test cases

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:

Rules for test case restriction

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 (often appearing combinations) are tested.

Test cases including **boundary values** or boundary value combinations are preferred.

Combine every representative of one equivalence class with every representative of other equivalence classes (i.e. pair wise combination instead of complete combination).
Ensure as minimum criteria that every representative of an equivalence class appears in at least one test case .
Representatives of invalid equivalence classes should not be combined with representatives of other invalid equivalence classes.

Test invalid values separately

Invalid equivalence classes are not combined in a "multiplicative way". An incorrect value should only be combined with "correct" 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, detect masking may result and only one of the possible exceptions is actually triggered and tested. On appearance of a failure, it is not obvious, which of the incorrect values has triggered the effect. This leads to extra expenses for analysis.

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 chapter 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; chapter 7.1.2).

Tab. 5–5 Valid and invalid equivalence classes of the parameter of the function

Parameter	Equivalence classes
baseprice	vEC_{11}: [MIN_DOUBLE, … , MAX_DOUBLE] iEC_{11}: NaN
specialprice	vEC_{21}: [MIN_DOUBLE, … , MAX_DOUBLE] iEC_{21}: NaN
extraprice	vEC_{31}: [MIN_DOUBLE, … , MAX_DOUBLE] iEC_{31}: NaN
extras	vEC_{41}: [MIN_INT, … , MAX_INT] iEC_{41}: NaN
discount	vEC_{51}: [MIN_DOUBLE, … , MAX_DOUBLE] iEC_{51}: NaN

Step 2: Refine the equivalence classes based on the specification

In order to further subdivide these equivalence classes, information about the functionality is needed. The functional specification delivers this information (chapter 3.2.3.). From this specification the following conclusions relevant for testing can be drawn:

> Parameters 1 to 3 are (vehicle-) prices. 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[37]. The specification does not define an upper limit for this data element.
> 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 of 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 during this analysis, partitioning them into subclasses. The more detailed the equivalence classes are the more comprehensive the test becomes. The class partition is complete when all conditions in the specification, as well as conditions from the tester's knowledge are incorporated.

37. Floating point numbers belong to the equivalence class NaN, see the example for equivalence partition for integer numbers.

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

Tab. 5–6
Further partitioning of the equivalence classes of the parameter of the function `Calculate_price()` *with representatives*

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. Any value of an equivalence class can be used, according to equivalence class theory. In practice perfect decomposition is seldom done. Because of 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[38]. Therefore, the choice of representative values for the test must be made with care. 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 borne in mind.

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 the table above.

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

Step 4: Combine the test cases

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

Tab. 5-7 *Further partitioning of the equivalence classes of the parameter of the function `Calculate_price()` with representatives*

	parameter					
Test case	baseprice	special-price	extraprice	extras	discount	result
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

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 feature only one valid equivalence class, only 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 should be identified for every test case. For the negative tests this is easy: The expected result is the error code or message generated by the test object. For the "valid" tests, the expected outcome must be calculated (for example by using a spreadsheet).

Definition of the Test Completion Criteria

A test completion criterion for the test by equivalence class partitioning can be defined as the percentage of executed equivalence classes in comparison to the total number of specified equivalence classes (→equivalence partition coverage):

EC-coverage = (number of tested EC / total number of EC) * 100 %

Let us assume that 18 equivalence classes have been defined, like in our example, but that only 15 have been executed in the chosen test cases, Then the equivalence class coverage is 83 %.

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

Example: Equivalence class coverage

In the total of 14 test cases (table 5-7), all 18 equivalence classes are contained with at least one representative each. Thus, executing all 14 test cases gives a 100 % equivalence class coverage. If the last three test cases are left out because of time reasons, i.e. only 11 instead of 14 test cases are executed, all thee invalid equivalence classes for the parameter `discount` are not tested and the coverage will be 15/18, i.e. 83,33 %.

Degree of coverage defines test comprehensiveness

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

If, in the example above, 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. Test coverage is a measurable criterion for finishing testing.

The example also shows the criticality of the identification of the equivalence classes. If not all the equivalence classes have been identified, then fewer values will be chosen for designing test cases, fewer test cases will result. A high coverage is achieved but it has been calculated based on a wrong total number of equivalence classes. The alleged good result does not reflect the actual test intensity. Test case identification using equivalence class partitioning is only as good as the analysis of the requirements and the following building of the equivalence classes.

The value of the technique

Equivalence class partitioning contributes to a complete test where specified conditions and restrictions are not overlooked. The method also minimizes the generation of unnecessary test cases. Such test cases are the ones having data from the same equivalence classes and therefore resulting in equal behavior of the test object.

Equivalence classes cannot only be determined 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 system-, integration- and component testing.

However, only single input or output conditions are considered, while possible dependencies or interactions between conditions are ignored. If they are considered, this is very expensive, but can be done through further partitioning of the equivalence classes and specifying according combinations. This is also called "domain analysis".

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

5.1.2 Boundary Value Analysis

Reasonable addition

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

Boundary value analysis checks the "border" of the equivalence classes. On every border, the exact boundary value and both nearest adjacent values (inside and outside the equivalence class) are tested. Thereby 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 another equivalence class. In such cases it can be sufficient to test the boundary with two values: One value, which is just inside of an equivalence class and one value, which is just outside.

Example: Boundary values for bonus

For paying the bonus (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: vEC_3: 5 < x <= 8 and vEC_4: x > 8. For testing the common boundary of the two equivalence classes, the values 8 and 9 can be chosen The value 8 lies in vEC_3 and is the largest possible value in that equivalence class. The value 9 is the least possible value in vEC_4 .The values 7 and 10 do not give any more information because they are further inside their corresponding equivalence classes. Thus, when are the values 8 and 9 sufficient and when should we additionally use the value 7?

It can help to look at the implementation. The program will probably contain the instruction `if (x>8)`. Which wrong implementation of this condition can be found by which test cases? The test values 7, 8 and 9 generate the truth-values `false`, `false` and `true` in the if-statement and the corresponding program parts are executed. Test value 7 does not seem to add any value because test value 8 already generates the truth-value `false`. Wrong implementation of the statement `if (x>=8)` leads to the truth values `false`, `true` and `true`. Even here, a test with the value 7 does not lead to any new results and can thus be omitted. Only a wrong implementation of `if (x<>8)` and the truth values `true`, `false` and `true` can only be found with test case value 7. The values 8 and 9 deliver the expected results or the same ones as with the correct implementation.

Hint: Wrong implementation of the instruction in `if (x<8)` with `true`, `false` and `false` and in `if (x<=8)` with `true`, `true` and `false` always result in two differences between actual and expected result and can be found by test cases with the values 8 and 9.

As the equivalence classes vEC_3 and vEC_4 are neighbor classes, a test with the input data 7 and 8 must lead to the specified output of vEC_3 (75 % bonus). A wrong implementation of a logic query thus leads to wrong results and will be found.

It should be decided when a test with only two values is considered enough and when it is interesting to test the boundary with three values. The wrong query in the example program, implemented as (`if (x<>8)`) can be found in a code review as it does not check the boundary of a value area (`if (x>8)`), but instead checks if two values are unequal. However, this defect can easily be overlooked. Only with a boundary value test with three values, all possible wrong implementations of boundary conditions are found.

Example: Integer input

For the above example of the test of an integer input value, five new test cases result, thus there will be a total of twelve test cases with the following test 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 EC_1: `[MIN_INT, _ 0[`. This test case also verifies the smallest deviation from the lower boundary (`0`) of the equivalence class EC_2: `[0, _, MAX_INT]`. The value lies outside this equivalence class. Just notice 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 twelve 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.

Is the test cost justified?

Here too, we have to decide if the test cost is justified and every boundary with the adjacent values each 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 test cases with the maximum and the minimum value inside the equivalence class are already chosen in some test case. In the example these values are `MIN_INT` +1 and 1, `MAX_INT`-1.

Boundaries not existing for sets

For the example with the input data element "traveler" given above no boundaries for the input domain can be found. The input data type is discrete, i.e. a set of the six elements (child, teenager, adult, student, person on welfare, retired person). Boundaries cannot be identified here. A possible order by age cannot be defined clearly because the person on welfare for instance might have any age.

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.

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 value test for *`calculate_price()`*

The following table lists the boundary values for the valid equivalence classes for verification of the function `calculate_price()`:

Tab. 5–8 *Boundaries of the parameters of the function `calculate_price()`*

Parameter	Lower boundary value [Equivalence class] Upper boundary value
baseprice	0-δ[a], [0, 0+δ, ..., MAX_DOUBLE-δ, MAX_DOUBLE], MAX_DOUBLE+δ
specialprice	Same values as baseprice
extraprice	Same values as baseprice
discount	0-δ, [0, 0+δ, ..., 100-δ, 100], 100+δ

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 invalid 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 representatives must be used for new test cases.

```
baseprice:      0.00, 0.01[39], 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 5-9).

The expected results of a boundary value test are not always easy to derive from the specification. The experienced tester must thus define reasonable expected results:

	Parameter					
Testcase	**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
...						

Tab. 5-9
Further test cases for the function `calculate_price()`

Test case 15 verifies all valid lower boundaries of equivalence classes of the parameters of `calculate_price()`. The test case seems not to be very realistic[40]. This is because of the imprecise specification of the functionality, where no lower and upper boundaries are specified (see below).[41]

Test case 16 is analogous to test case 15, but here we test the precision of the calculation[42].

Test case 17 combines the next boundaries from the table above. The expected result is rather speculative with a discount of 99.99 %. A look into

39. For the test cases, 0.01 was assumed to be precise enough.
40. 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.
41. The dependence between the number of extras and extra price (if no extras are given, there should not be a price given) cannot be checked through equivalence partitioning or boundary value analysis. In order to do this cause-effect analysis [Myers 79] must be used.
42. In order to exactly check the rounding precision, values like for example 0.005 are needed.

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 results when the values of the other parameters are set to 0.00, in order to check if maximal value 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.

Early thinking of testing pays off

The example shows the detrimental effect of imprecise specifications. If the testers discusses with the customer before determining the test cases, and the value ranges of the parameters can be specified more precisely, and the test can be cheaper. This is shown here, as a short example.

The customer has given the following information:

The base price is between 10000 and 150000.
The extra prices for the extra items is between 800 and 3500.
There are maximum 25 additional extras, whose prices are between 50 and 750.
The dealer discount is maximum 25 %.

After specifying the equivalence classes, the following valid boundaries result:

```
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[43]
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. For values outside the valid equivalence classes we need one test case each. The following values must be used for these:

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

43. The maximum price for extra items cannot be specified exactly, because the dependence between number of extras and total price cannot be used. We used the value 25 * 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 specific specification results in less test cases and an easier 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 above, it must be decided if it is sufficient to test a boundary with two instead of three test data. In the following hints we assume that two test vales are sufficient, because a code review has been done and possible totally wrong checks have been found.

Hint on test case design by boundary analysis

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[44].

An input file has a restricted number of data records, between 1 and 100. The test values should be 1, 100 and 0, 101.

If the *output* domains serve as the basis then the analysis can be done as follows: 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 take a certain effort to identify the respective input test data to achieve exactly the required outputs. Generating the invalid outputs can even be impossible, but attempting to do it may find defects.

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 an 5 data values.

For ordered sets the first and last element is of special interest for the test.

If complex data structures are given as in- or output, for instance an empty list or zero matrixes can be considered a boundary value.

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.

In addition, extremely large data structures, lists, tables etc. should be chosen, for example ones that 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 element are of interest, as they often show failures due to wrong programming (*Off-by-one problem*).

44. The accuracy to be chosen depends on the specified problem.

Definition of the Test Completion Criteria

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

BV-Coverage = (number of tested BV / total number of BV) * 100 %

Notice though that not only the boundary values but also the according adjacent values above and below the boundary must be counted. However, only unequal values are used for the calculation. Overlapping values of adjacent equivalence classes are counted as one boundary value because only one test case with the respective input test value is possible.

The value of the technique

In combination with Equivalence class partitioning

Boundary value analysis should be done together with equivalence class partitioning because faults are discovered more often at the boundaries of the equivalence classes than far inside the classes. Both techniques can be combined easily but still allow enough degrees of freedom in selecting the concrete test data.

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

5.1.3 State Transition Testing

Consider history

In many cases, not only the current input but also the history of execution or events or inputs influences the outputs and how the test object will behave. To illustrate the dependence on history, →state diagrams are used. They are the basis for designing the test (→state transition testing).

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

Definition finite state machine

[Beizer 95] defines a finite state machine as follows: "An abstract machine (e.g., program, logic circuit, car's transmission) 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)." The comments given in parenthesis define the notation in a state graph or state transition diagram. A state diagram is a diagram that depicts the states that a system or component can assume, and shows the events or circumstances that cause or result from a change from one state to another [IEEE 610.12].

Example: Stack

The popular example of a stack (figure 5-3) is supposed to clarify the circumstances. The stack, for example a dish stack in a heating device can adopt three different states: an empty stack, a filled stack and a full stack.

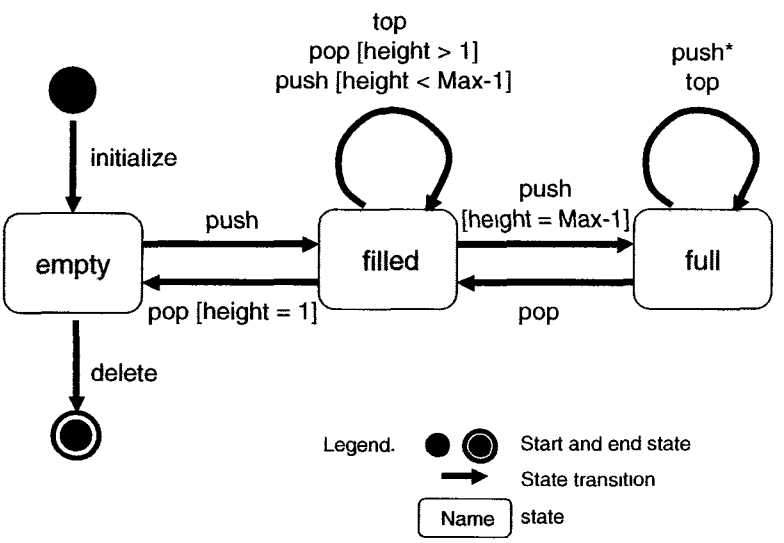


Fig. 5–3
State diagram of a stack

The stack is "empty" after initializing (init) where the maximum height (Max) is defined (current height = 0). By adding an element to the stack (calling push, the state changes into "filled" and the current height is incremented. In this state further elements can be added (push, increment height) as well as withdrawn (pop, decrement height). The uppermost element can also be displayed (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. If one element is removed (pop) while the stack is in the state "full", the state is changed back from "full" to "filled". A state 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 state "empty".

Depending on the specification, it can be defined which functions (push, pop, top, ...) can be called at which state of the stack. It must still be clarified what shall happen when an element is added to a full stack (push˚). The func-

tion must perform differently than in case of a just filled stack. The functions must behave differently depending on the state of the stack. Thus, the state of the test object is a decisive part and must be considered when testing.

A possible test case

Example: A stack accepting strings (type: string) shall be tested. A possible test case with pre- and postcondition is the following:

Pre-condition:	stack is initialized; state is "empty"
Input:	push ("hello")
Expected result:	stack contains "hello"
Post-condition:	state of the stack is "filled"

Further functions of the stack (display of the current height, display of the maximum height, query if the stack is "empty", ...) are not included in this example because they do not cause any change of the state.

Test object for 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 history leads to differing behavior, a state transition test has to be applied.

Further test cases for the stack example

For the state transition test, different levels of test intensity can be defined. A minimum requirement is to reach all possible states. In the given stack example these states are empty, filled and full[45]. With an assumed maximum height of 4 all three states are reached after calling the following functions:

Test case 1[46]: init [empty], push [filled], push, push, push [full]

Not even all of 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: init [empty], push [filled], top, pop [empty], delete

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

Test criteria

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

In order to identify the necessary test cases, the finite state machine is transformed into a transition tree, which includes certain sequences of transitions ([Chow 78]). The cyclic state transition diagram with

45. To keep the test effort small the maximum height of the stack should be chosen not too high because the push-function must be called a corresponding number of times to reach the state "full".
46. The following test cases are simplified to keep them manageable.

potentially infinite sequences of states changes to a transition tree, which corresponds to a representative number of states without cycles. In doing this translation, all states must be reached and all transitions of the transition diagram must appear.

The transition tree is build from a transition diagram in the following 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. Process step 2 is repeated for every leaf in the tree (every newly added node) until one of the 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 pass of a cycle in the transition diagram.
   The corresponding state is a final state and has therefore no further transitions to be considered.

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

Eight different paths can be produced from the root to each of the end nodes. Each of the paths represents a test case, i.e. 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.

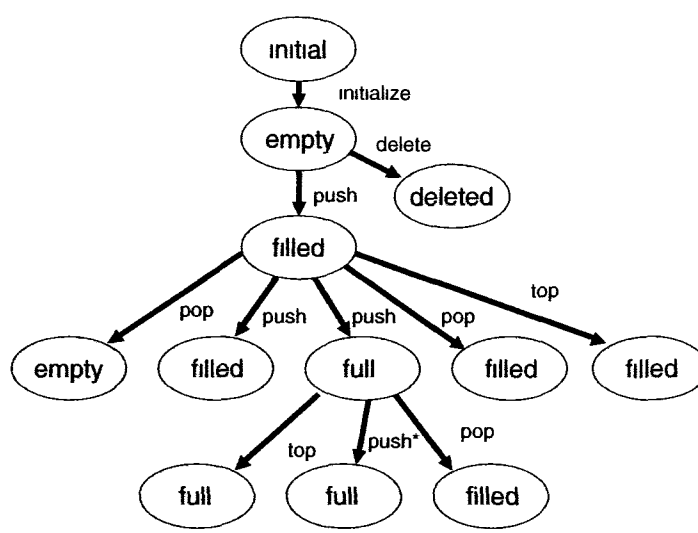


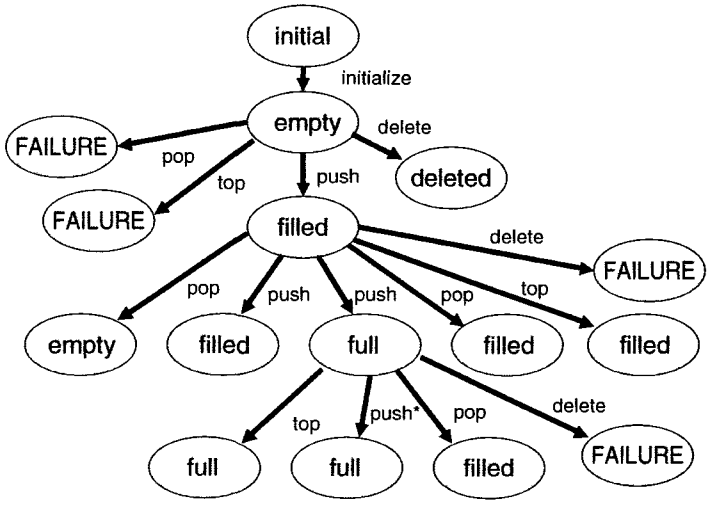
Fig. 5-4
Transition tree for the stack-example

Wrong usage of the functions

In addition to this, the reaction of the state machine for wrong usage has to be checked. This means functions are called in states in which they are not supposed to be called (e.g., to delete the stack while in "full" state). This is a test of robustness to verify how the test object reacts upon incorrect use. It is tested if unexpected transitions may appear. The test can be seen as an analogy to the test of unexpected input values.

The transition tree should be extended by including a branch for every function from every node. This means that from every state all the functions should be executed or at least attempted to be executed (figure 5-5).

Fig. 5–5
Transition tree for the test for robustness



State transition testing is also a good technique for system test. For example, the test of a Graphical User Interface (GUI) can be designed this way. The Graphical User Interface usually consists of a set of screens and user controls, such as menus and dialog boxes. Between those the user can switch back and forth (menu choices, "OK"-Button, etc.). If screens and user controls are seen as states and input reactions as state transitions then the Graphical User Interface can be modeled as a finite state machine. Appropriate test cases and the test coverage can be identified by the technique of state transition based tests given above.

For the test of the *DreamCar* – GUI this can look like the following figure:

Example: Test of the DreamCar-GUI

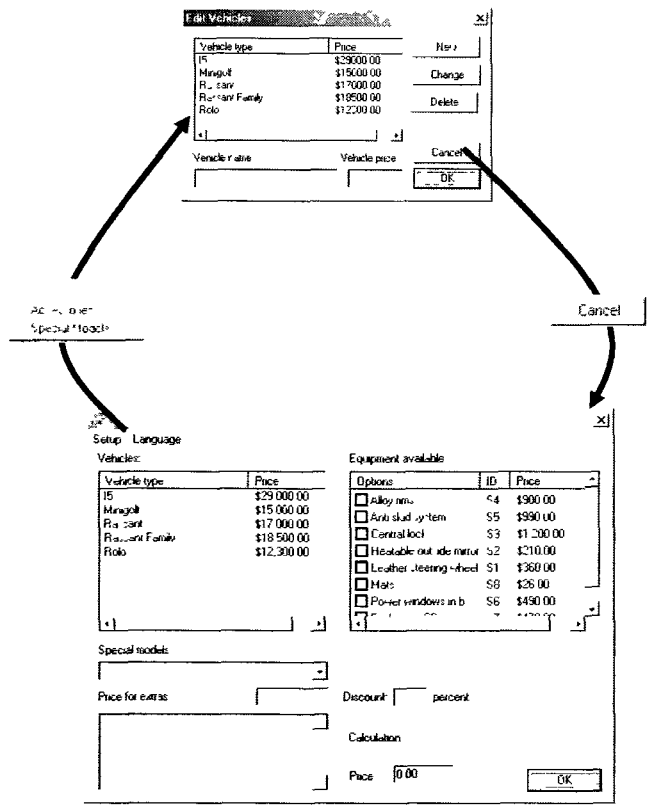


Fig. 5–6
GUI-navigation as state graph

The test starts at the *DreamCar* main screen (state 1). The action[47] "Option/vehicles" triggers the transition into the dialog "Edit vehicle" (state 2). The action "Cancel" ends this dialog and the return to state 1 occurs. Inside a state we can then use tests, which do not change the state. These local tests then verify the actual functionality of the accessed screen. Like this, navigation through arbitrarily complex chains of dialogs can be modeled. The state diagram of the GUI ensures that all dialogs are included and verified in the test.

47. The two-staged menu choice is seen here as on action.

Test Cases

For a complete definition of 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.

Again, for each expected 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 hidden deep in the test object. Thus, the verification and evaluation of each test case can be very expensive.

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

Definition of the Test Completion Criteria

Criteria for test intensity and for completion can also be defined for the 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 the actually executed test requirements to the possible ones analog to the earlier described coverage measures.

Higher-level criteria

For highly critical applications even further intensified state transition test completion criteria can be declared:

All combination of transitions.
All transitions in any order with all possible states, multiple instances in succession, too.

But achieving a sufficient coverage is often not possible due to the large number of necessary test cases. A limitation of the number of combinations or sequences that must be verified may then be reasonable.

The value of the technique

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

Especially useful for test of OO systems

In object-oriented systems objects can have different states. The appropriate methods to manipulate the objects must then react according to the different states. State transition test is of greater importance for object-oriented testing because it takes into account the special aspects of the object orientation.

5.1.4 Cause-Effect Graphing and Decision Table Technique

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

Cause-effect graphing

[Myers 79] describes a technique that uses the dependencies for identification of the test cases: →Cause-effect graphing. The logical relations between the causes and their effects in a component or a system are displayed in a so-called →cause-effect graph. It must be possible to find the causes and effects from the specification. Every cause is described as a condition that consists of input conditions (or combinations of those). The conditions are connected with logical operators (e.g. AND, OR and NOT). A condition and therefore a cause can be `true` or `false`. The effects are treated in the same way and noted in the graph (figure 5-7).

Example: Cause-effect graph analysis for an ATM

An example, getting money at an automated teller machine (ATM), shall clarify how to prepare a cause-effect graph. In order to get money from the machine, the following conditions must be fulfilled[48]:

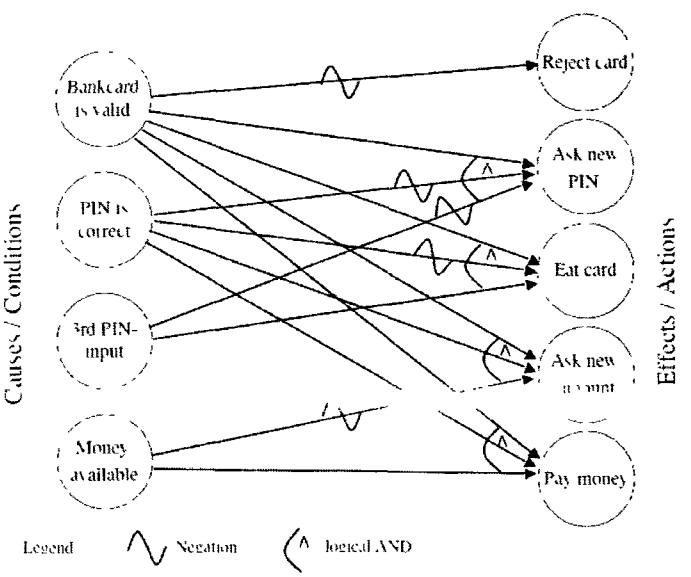
The bankcard is valid.
The PIN must be correctly entered.
The maximum number of PIN inputs is three.
There is money (in the machine and the account).

The following actions are possible at the machine:

Reject card.
Ask for another PIN input.
"Eat" the card.
Ask for another amount.
Pay the amount of money.

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

Fig. 5–7
Cause-effect graph of the ATM



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

The graph must be transformed into a →decision table from which the test cases can be taken. The steps to transform a graph into a test table are the following:

48. Hint: This is not a complete description of a real automated teller machine, just an example to illustrate the technique.

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

Test with decision tables

The test based on decision tables has the objective to design tests for execution "interesting" combinations of inputs. Interesting in the sense that possible failures can be detected. Besides the causes and effects intermediate results 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 is a test case, i.e. the combination of conditions and the expected effects or outputs for this combination.

In the least optimized case, every combination of causes is considered as a 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 with "yes" or "no". Each cause and effect should occur at least once with "yes" and "no" in the table.

From a decision table a decision tree may be derived. The decision tree may be used analogous to the transition tree in state transition testing. 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, which determines the further path depending on its truth-value.

As there are four conditions (from "bank card is valid" to "there is no money"), there are, theoretically, 16 (2^4) possible combinations. However, dependencies are not taken into account here. For example, if the bankcard is invalid, the other conditions are not interesting, as 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 any more. As there are dependencies between the inputs and the results (actions, outputs), the following optimized decision table shows the result (table 5-10).

Tab. 5-10 Optimized decision table for the ATM

Condition / Cause	1	2	3	4	5
Bankcard is valid	N	Y	Y	Y	Y
PIN is correct	-	N	N	Y	Y
3 incorrect PIN	-	N (exit)	Y	-	-
Money available	-	-	-	N	Y
Effect / Action					
Reject card	Y	N	N	N	N
Ask new PIN	N	Y	N	N	N
Eat card	N	N	Y	N	N
Ask new amount	N	N	N	Y	N
Pay money	N	N	N	N	Y

Every column of this table should 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 only output, if the card is valid, the PIN is correct after maximum three tries and there is money available.

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

Test cases

Every column is a test case

In a decision table the conditions and dependencies for the inputs and the corresponding outputs and results for this combination of inputs can be read directly from every column. The table defines logical test cases. They must be fed with concrete data values in order to execute them, and necessary pre-and postconditions must be annotated.

Definition of the Test Completion Criteria

Simple criteria for test completion

As with the previous methods criteria for test completion can be defined relatively easily. A minimum requirement is to execute every column in the 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 reveal combinations, which are not included when using other test case design techniques. However, errors can result from optimization of the decision table, for example when the input and condition combinations to be considered are left out.

As mentioned above, the graph and the table may grow very fast and loose readability when the number of conditions and dependent actions increases. Without adequate support by tools, the technique is not easily applicable.

5.1.5 Use Case Testing

UML widely used

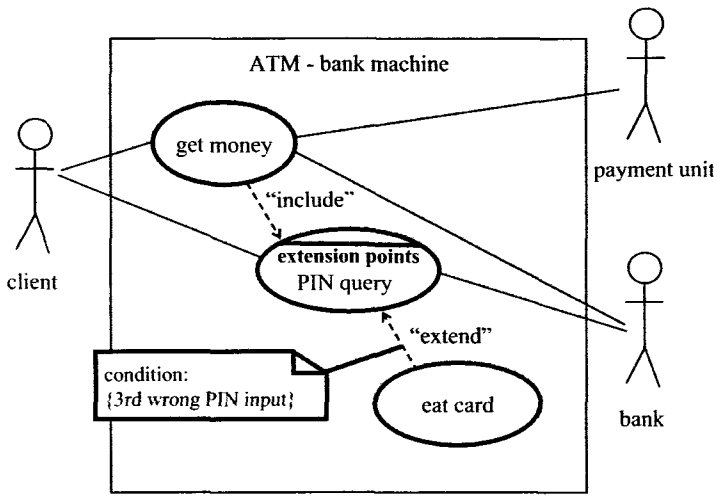
With the increasing use of object-oriented methods for software development the Unified Modeling Language (UML) ([URL: UML]) is more and more used. UML defines more than ten graphical notations, which may be used in software development, not only if it is object-oriented. There exist quite a number of (research) results and approaches to directly derive test cases from UML-diagrams and to generate them more or less automatically. This chapter will only describe the use of use cases or use case diagrams.

Identification of the requirements

In order to detect requirements, use cases or business cases are described. These are then compiled into use case diagrams. The diagrams serve the purpose of defining requirements on a relatively abstract level and describe typical user-system interactions. Figure 5-8 shows a use case diagram for a part of the dialog when using an ATM for getting 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 involved, "extend" connections can lead to extensions of a use cases under certain conditions at a certain point (*extension point*). Thus, the "extend" conditions is not always executed, there are alternatives.

Fig. 5–8
Use case diagram for ATM



Showing an external view

Use case diagrams mainly serve to show the external view of a system. It shall explain the external view of the system from the viewpoint of the user or 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, which are not further discussed here.

Pre- and postconditions

For every use case, there exist certain preconditions, which must be fulfilled if it shall be possible to execute the use case. A precondition for getting money at the ATM is, for example, that the bankcard is valid. After executing a use case there exist 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 checked if the money is available. Pre- and postconditions are also applicable for the flow of use cases in a diagram, i.e. the path through the diagram.

Useful for system and acceptance testing

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 useful for system- and acceptance testing. If the diagrams are used to model the interactions between different subsystems, test cases can also be derived for integration testing.

Typical system use is tested

The diagrams show the "normal" or typical or probable flows and often their alternatives. Thus, the use case based test checks typical use of a system. It is certainly especially important for acceptance of a system that it runs relatively stable in "normal" use. Thus use case based testing has a high relevance for the customer and therefore also for the developer and tester.

Test Cases

Every use case has a purpose and shall achieve a certain result. There can occur events, which lead to further alternatives or activities. After the execution there are postconditions. All this information is necessary for determining the test cases and must be available:

Start situation and preconditions
Possible other conditions
Expected results
Postconditions

However, the concrete input data and results for the individual test cases cannot be derived directly for the use cases. Analysis of the concrete conditions for the individual input and output data is necessary. However, each alternative contained in the diagram ("extend"-relation) must be covered by a test case.

Definition of the test completion criteria

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

The value of the technique

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

5.1.6 Further Black Box Techniques

This chapter contained a description of some black box techniques but this is far from complete. Below, a few more practical techniques are described shortly, in order to give hints about their selection. Further techniques can be found in [Myers 79], [Beizer 90], [Beizer 95], and [Pol 98]. The following section describes other techniques briefly to offer assistance with their selection.

Syntax test

→Syntax testing describes a technique for identification of the test cases that may be applied if a formal specification of the syntax of the inputs is available. This may be the case for testing interpreters of command languages, compilers and protocol analyzers. The rules of the syntax definition are 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 can be used for the selection of test values. This ensures the derivation of test cases that are preferably close to reality and makes 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. Its common understanding is that of a "quick and dirty" test which is primarily aimed at verifying a 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. The main interesting outcome is a crash or serious misbehavior of the test object. A test oracle is not used and this contributes to making this test cheap and easy. The term is derived from testing electrical circuits where short circuits lead to smoke rising. A smoke test is often used to decide if the test object is mature enough to be tested further by the more comprehensive test techniques. A further use of smoke tests is the first and fast test of software updates.

5.1.7 General Discussion of the Black Box Technique

Faults in specification not detected

The basis of all black box techniques is the requirements or the specification of the system or its components and their collaboration. Black box testing will, when applied slavishly, not 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 program under execution. The test object executes, as the requirements or specification require it, even when they are wrong. If the tester is critical towards the requirements or specifications and uses "common sense", wrong requirements can be found during test design. Otherwise, to find inconsistencies and problems in the specifications, reviews must be used (chapter 4.1.2.).

Not required functionality is not detected

In addition, black box testing cannot reveal extra functionality that exceeds the specification. (Such extra functionality is often the cause of security problems). Such additional functions are neither specified nor required by the client. Test cases that execute those additional functions are, if at all, performed by pure chance. The coverage crite-

ria, which serve as condition for test completion, are exclusively identified on basis of the specification or requirements. They are not identified on the basis of unmentioned or just assumed functions.

Verification of the functionality

The center of attention for all black box techniques is the verification of the functionality of the test object. It is definitely not controversial that the correct working of a software system has the highest priority. Thus, black box techniques should always be applied.

5.2 White Box Testing Techniques

Code-based testing techniques

The basis for white box techniques is the source code of the test object. These techniques are therefore often called →code-based testing techniques or structural testing techniques. The source code must be available and in certain cases it must be possible to manipulate it, i.e. to add code.

All code should be executed

The generic idea of the white box techniques is to execute every part of code of the test object at least once. Flow-oriented test cases are identified analyzing the program logic and then executed. However, the expected results should be determined using the requirements or specifications, not the code. This is done in order to decide if execution resulted in a failure.

The focus of examination of a white box technique can for example be the statements of the test object. Primary goal of the technique is then to achieve a previously defined coverage of the statements while testing, i.e., to execute as many as possible statements in the program.

There exist the following basic white box test case design techniques:

- →statement coverage
- →branch coverage
- Test of conditions
  - branch condition testing[49] (→branch condition coverage)
  - →branch condition combination testing
  - condition determination testing
- path coverage

The following sections describe these techniques in more detail.

49 A related technique is called "modified condition decision coverage" (MCDC) For the differences between both techniques see the glossary

5.2.1 Statement Coverage

Control flow graph necessary

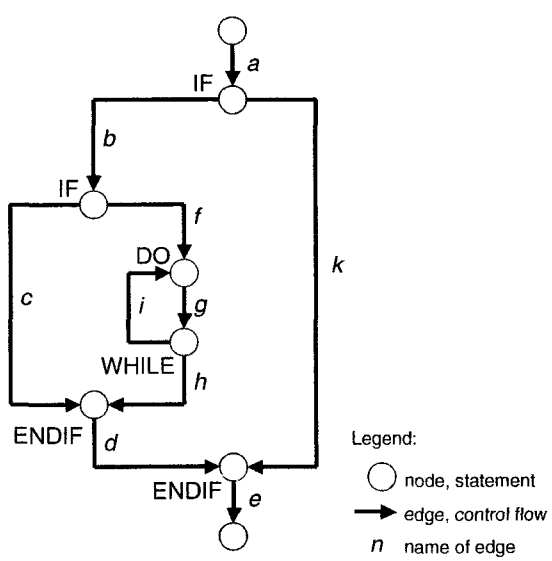
This analysis focuses on each statement of the test object. The Test cases shall execute a predefined minimum quota or even all statements of the test object. The first step is to translate the source code into a control flow graph. The graph makes it easier to specify in detail the control elements that must be covered. In the graph, the statements are represented as nodes and the control flow between the statements is represented as edges (connections). If sequences of unconditional statements appear in the program fragment then 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) have more than one edge going out from them.

After execution of the test cases it must be verified which of the statements have been executed (chapter 5.2.1). When the previously defined coverage level has been achieved, the test is considered to be sufficient and will therefore be terminated.

Example

The following example should clarify the proceeding. A very simple program fragment is chosen for this example. It only consists of two decisions and one loop (figure 5-9).

Fig. 5-9
Control flow of a program fragment



Test Cases

Coverage of the edges of the control flow

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

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

One test cases is enough

After traversing the edges this way, all statements have been executed once. Other combinations of edges of the graph surely can also achieve complete coverage. But the cost of testing should always be minimized, that means reaching the goal with the least possible number of test cases.

The expected results and the expected behavior of the test object should be identified in advance from the specification. After execution, the expected and actual results and the behavior of the test object must be compared to detect any difference or failure.

Definition of the Test Completion Criteria

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

Statement coverage =
(number of executed statements / total number of statements) *100 %

C0-measure

Statement coverage is also known as C0-coverage. It is a very weak criterion. However, sometimes 100 % statement coverage is difficult to achieve, for instance, if exception conditions appear in the program, which 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 cases then this may be an indication for non-reachable source code (dead statements).

Empty ELSE-parts are not considered

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

Coverage is measured using automated tools (chapter 7.1.4).

5.2.2 Branch Coverage

A more advanced criterion for white box testing is →branch[50] coverage of the control flow graph; i.e. the edges in the graph are the center of attention. Not the execution of each statement is considered but the execution of decisions. The result of the decision decides which statement is executed next. Testing should make sure every decision is executed with both possible outcomes (TRUE and FALSE – →decision coverage is another name for this criterion).

Empty ELSE-parts are considered

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, i.e. both THEN and ELSE in the IF-statement, all possibilities for the CASE-statement and for loops both execution of the loop body and the fall-through case, i.e., the bypassing of the loop body and a return to the beginning of the loop.

Test Cases

Additional test cases necessary

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. Additionally, three further test cases are necessary:

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

Connection (edge) coverage of the control flow graph

Together all four test cases result in a 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.

Some edges have been executed more than once. This seems to be redundant. However, 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.

---

50. See the glossary for the definition of branch.

For each test case, besides the pre- and postcondition, 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 of the branches have been executed in which test case. This helps to find faults, especially missing code in empty branches.

Definition of the Test Completion Criteria

Analogous to the statement coverage the degree of coverage for branch coverage is defined as follows:

Branch coverage =
(number of executed branches / total number of branches) * 100 %

C1-measure

Branch coverage is also called C1-coverage. The calculation just counts if a branch has been executed. The frequency of execution is not relevant. In the example the edges a and e are each passed four 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, i.e.

$$9/10 * 100\ \% = 90\ \%.$$

For comparison: 100 % statement coverage has already been reached after the first test case.

Depending on the criticality of the test object and depending on the expected failure risk the test completion criterion can be defined differently. For instance 85 % branch coverage can be sufficient for a component of one project whereas 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

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. 100 % branch coverage 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 combination of single branches is required.

Hint

A branch coverage of 100 % should be aimed for.
Only if, in addition to all statements every possible branch of the control flow is considered during test execution, then the test can be categorized as sufficient.

Inadequate for object-oriented systems

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

5.2.3 Test of Conditions

Considering the complexity of combined conditions

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 is chosen and accordingly which statement is the next to be executed in the program. If a decision is based on several (part) conditions connected by logical operators then the complexity of the condition should be considered in the test. The following sections describe different requirements and thence degrees of test intensity under consideration of the composed conditions.

Branch condition testing

The goal of the branch condition testing is that each →atomic (partial) condition in the test shall adopt the values `true` and `false`.

Definition atomic condition part

An atomic part of a condition is a condition that has no logical operators like AND, OR and NOT but at the most includes relation symbols like ">" or "=". 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 composed condition is: `x > 3 OR y < 5`. The condition consists of two atomic partial conditions (`x>3; y<5`) connected by the logical operator `OR`. The goal of the branch condition testing is that every atomic part of conditions is evaluated once for each of the logical values. The test data x=6 and y=8 result in the logical value `true` for the first part of condition (`x>3`) and the logical value `false` for the second part of the conditions (`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 part of condition and `true` for the second part of condition. The value of the complete condition results in `true` again (`false OR true = true`). Both parts of the condition have each resulted in both logical values. The result of the complete condition, however, is equal for both combinations.

Weak criterion

Branch condition testing 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.

Branch Condition Combination Testing

All combinations of the logical values

Branch condition combination testing, also called multiple-condition coverage [Myers 79] 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

For the example above four combinations of test cases are possible with the test data from above for the two atomic parts of 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)

Branch condition combination testing includes statement and branch coverage

The complete condition gives both logical values as results. Thus, branch condition combination testing meets the criteria of statement as well as branch coverage. It is a more comprehensive criterion that also takes into account the complexity of composed conditions. But it is a very expensive technique because a growing number of atomic conditions make the number of possible combinations grow exponentially (to 2^n with n atomic parts of conditions).

All combinations are not always possible

A problem results from the fact that not always all combinations can be implemented by test data.

Example for not feasible combinations of condition parts

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

x=4: 3<=x (T), x<5 (T), 3<=x AND x<5 (T)
x=8: 3<=x (T), x<5 (F), 3<=x AND x<5 (F)
x=1: 3<=x (F), x<5 (T), 3<=x AND x<5 (F)
x=?: 3<=x (F), x<5 (F), combination not possible because the value x shall be smaller than 3 and greater or equal to 5 at the same time.

Condition Determination Testing

Restriction of the combinations

Condition determination testing eliminates the problems that just have been discussed. Not all combinations must be considered but rather every possible combination of logical values where the modification of the logical value of an atomic condition can change the logical value of the whole. Stated in another way, for a test case every atomic condition has a meaningful impact on the result. Test cases where the result does not depend on a change of an atomic condition need not be designed.

Continuation of the example

For clarification we revisit the example with the two atomic condition parts (`x>3, 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)

Changing a partial condition without changing the result

For the first combination the following applies: If the logical value is calculated wrong for the first condition part (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 condition part.

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

If the logical value of the first condition part in the second test case is calculated wrongly as `false`, then the result value of the total condition changes from `true` (T) to `false` (F). A failure then becomes visible because the value of the complete condition has also changed. The same applies for the second condition part 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 conditions 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. The number of test cases is considerably smaller compared to the branch condition combination testing. The amount lies between n+1 and 2n with n = number of the Boolean operands of the condition.

Test Cases

For designing the test cases, it must be considered which input data lead to which result of the condition or condition part 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 or not.

Hint

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

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

Definition of the Test Completion Criteria

Analogous to the previous techniques the proportion between the executed and all the required logical values of the condition (parts) can be calculated. This can serve as criteria for termination of the tests. For the techniques, which concentrate attention to the complexity of the conditions in the source code, it is reasonable to try to achieve a complete verification (100 % coverage). If there are no complex condition expressions, branch coverage can be seen as sufficient.

The Value of the Technique

Complex conditions are often defect-prone

If complex conditions are present in the source code, they must be tested intensively to uncover possible failures. Combinations of logical expressions are especially defect-prone. Thus, a comprehensive test is very important. Condition determination testing admittedly is a very expensive technique for test case design.

Hint

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 (chapter 4.1.2) in which the correctness is verified.

Excursion

A disadvantage of →condition coverage is that it checks Boolean expressions only inside a statement (for example, IF-statement). In the following example of a program fragment, it is not detected that the IF-condition is combined of multiple condition parts and that modified branch condition determination testing should be applied.

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

If all Boolean expressions that appear in the program are analyzed for construction of combined conditional test cases then this disadvantage can be prevented.

Compiler terminates evaluation of expressions

Another problem occurs in connection with measuring the coverage of condition parts. Some Compilers shortcut the evaluation of the Boolean expression as soon as the total result will not change any more. For instance, if the value FALSE has been detected for one of two condition parts of an AND-concatenation then the complete condition is FALSE regardless of what the second condition part will result in. Some compilers even change the order of the evaluation depending on the Boolean operators in order to receive the final result as fast as possible and to be able to disregard any other condition part. Test cases that are supposed to reach coverage of 100 % can be executed but because of the shortened evaluation, the coverage cannot be verified.

5.2.4 Path Coverage

All possible paths through the test object

Until now, test case determination focused on the statements or branches of the control flow as well as the complexity of conditions. If the test object includes loops or repetitions, the previous deliberations 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

Considering the control flow graph (figure 5-9), we try to clarify the term "path". The program fragment represented by the graph includes a loop. This DO-WHILE loop is executed at least once. In the WHILE-condition it is decided at the end of the loop whether the loop must be repeated, i.e. if a jump 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

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 is 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 chapter 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 like in loops for example, at which one branch leads back to the beginning of another branch.

Example: Statement and branch coverage VCR

In chapter 5.1.1 for the function `calculate_price ()` of the VSR-par-system *DreamCar* test cases from valid and invalid equivalence classes of the parameters have been chosen. In the following, test cases are evaluated by their ability to cover the source code, i.e. accordingly execute fragments of the method. 100 % coverage should be achieved in order to ensure that during test execution all branches have been passed at least once.

For better understanding the source code of the function from chapter 3.2.3 is displayed again:

```
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.

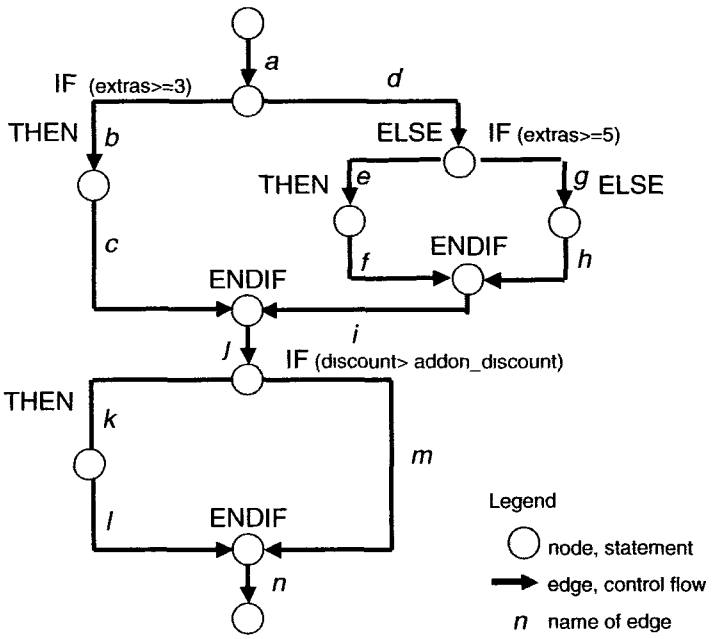


Fig. 5-10
Control flow graph of the function `calculate_price()`

In chapter 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 was supposed to be calculated (with five or more pieces of extra equipment).

In order to increase the coverage the following additional 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

The 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, i.e. the source code is analyzed in order to define further test cases. To reach the edges e and f the condition of the first condition (`extras >= 3`) must be `false` in order to execute the ELSE-part. In this ELSE-part the condition (`extras >= 5`) must be `true`. Therefore a value has to be found that meets the following condition:

```
¬(extras >=3) AND (extras >=5)
```

There is no such value and the missing edges can never be reached. Here is a defect in the source code.

Example: Relationship between the measures

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

All statements (nodes) are reached 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. A sequence might look as follows:

a, b, c, j, m, n

This additional sequence should have replaced the first sequence (test case) above. With the resulting three test cases, branch coverage of 100 % is achieved.

Further paths through the graph

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

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 with 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

If there are loops in the source code then every possible number of repetitions is counted as one possible path through the program fragment. It is obvious that 100 % path coverage is not feasible in a program as soon as the program is not trivial

5.2.5 Further White Box Techniques

There are a number of additional white box test techniques. However, these are not described here. This chapter described the most common techniques. Further techniques are explained in [Myers 79], [Beizer 90], and [Pol 98]. The following section describes one technique a little closer.

Data flow based techniques

A number of techniques uses the data flow through the test object as the basis for identifying the test cases. Primarily the data usages in the test object are verified. The use of every variable is analyzed. The definitions of variables and the read and write accesses of variables are distinguished. These techniques may find faults where a value given to a variable in one place leads to failure at another place it is used. Furthermore, it is analyzed if the value of the variable is used for calculation of another variable or for identification of the truth-value of a condition. By means of this information, different criteria in relation to the data flow can be defined. These should then be covered by test cases. A detailed description of the data flow based techniques can be found in [Clarke et al. 85] .

5.2.6 General Discussion of the White Box Technique

Determine test intensity

Basis for all white box techniques is the source code. Adequate test case design techniques can be chosen and applied depending on the complexity of the program structure. Considering the source code and the selected technique the intensity of the test is defined.

White box techniques are suited for the lower test levels. For example, it is not very reasonable to require coverage of single statements or branches at system test because system testing is not the right method to check single statements or conditions in the code.

Useful for lower test levels

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

'Not existing source code is not considered

5.2.7 Instrumentation and Tool Support

White box techniques require that different program fragments are executed and conditions get different logical values. In order to be able to evaluate the test, it must be determined which program fragments have already been executed and which fragments have not been executed yet. For that purpose, the test object has to be instrumented at strategic relevant spots of the 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 according program fragments. If a counter stayed zero during the test then the according program fragment has not been executed.

Determination of the executed program parts

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 (chapter 7.1.4). These tools are very important for white box testing because they increase the productivity and indirectly improve the quality of the test object.

Use tools

5.3 Intuitive and Experience Based Test Case Determination

Besides the methodical approach, intuitive determination of test cases should be performed. The systematically identified test cases may be complemented by intuitive test cases. Intuitive testing can uncover faults overlooked by systematic testing.

Basis of this method is the skill, experience and knowledge of the tester. The tester selects test cases to uncover expected problems and their symptoms. A more systematic approach for this cannot be described. The test cases base on the experience about where faults

Intuitive skill and experience of the tester

that have occurred in the past or the tester's assumptions 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.

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, there exist experiences with a new programming language in previous projects, it is reasonable to apply the failures found as well as their cause in using the programming language for designing the tests in the actual project. One technique for intuitive testing, exploratory testing, will be discussed in more detail in the following.

Exploratory Testing

If the documents, which form the basis for test design, are of very low quality or do not exist at all, so-called "exploratory testing" may 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 intuition and experience of the tester.

The approach in "exploratory testing"

The test activities in exploratory testing are executed nearly in "parallel". There is no application of a structured test process. 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. Few test cases are executed and their results are analyzed. Executing them, the "unknown" behavior of the test object will be determined further. Anything considered special and other information are then used to determine the next test cases. This way, step by step, 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 it there is time left.

"Test charter"

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

Why, with which goal, is the test run?
What is to be tested?
How, i.e. with which method, should be tested?
Which problems should be found?

Main features of exploratory testing

The generic ideas of exploratory testing are:

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 be.
The test is run against this model. The focus is to find further aspects and behavior of the program, which are still not part of the mental model or are differing from aspects found before.

Neither black box nor white box

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 basis for the considerations. Its range of application is rather in the higher test levels. In the lower ones usually sufficient information like source code or detailed specification is accessible for applying systematic techniques.

Not to be used as first or only technique

The intuitive test case determination should not be applied as the primary testing technique. Instead, this technique should be used to support and complete the choice of test cases through systematic testing techniques.

Test Cases

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

Example: Tester knowledge for the CarConfigurator

In case of the development project for the *CarConfigurator*, the testers are very familiar with the previous system. Many of them have tested this system as well. They know which 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 dealers). Employees from the companies' marketing department know for the business-process-based test which vehicles in which configurations are sold often and what 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.

Using all knowledge

The tester is supposed to use all her knowledge to find additional test cases. Naturally, the pre- and postconditions, the expected outcome and the expected behavior of the test object must be defined in advance for intuitive testing, too.

Hint

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 identified possible trouble and critical situations, additional test cases can be determined.

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

Definition of the Test Completion Criteria

The test exit criterion is not definable

Contrary to the systematic techniques a criterion for termination cannot be specified. If the above-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. However, they should only be used in addition to systematic techniques. The success and effectiveness of this approach depend very much on tester skill and intuition and their previous experience with applications like the test object and the used technologies. Such approaches can also contribute to find holes and errors in the risk analysis. If intuitive testing is executed in addition to systematic testing, hitherto no detected inconsistencies in the test specification 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 quite a number of techniques for testing of software.[51] The question is: When each of the techniques should be

51. There exist other techniques not described in this book. The reader should check further literature in case of need. This applies especially to integration testing, test of distributed applications and test of real time and embedded programs. Such techniques are part of the Advanced Level Tester Certification Scheme.

applied? The following gives advice and shows a reasonable procedure for answering this question. The general goal is to identify sufficiently different test cases, using whatever available method, in order to be able to find existing faults with a certain probability with as little effort as possible. The techniques for test design should therefore be chosen "appropriately".

However, before doing the work some factors should be checked, which have considerable influence on the selection or even prescribe the application of certain test methods. The selection of techniques should be based on different kinds of information:

The kind of test object
The complexity of the program text can vary considerably. Adequate test techniques should be chosen. If, for example, conditions in the program are combined from atomic subconditions, branch coverage is not sufficient. A suitable technique to check the conditions should be chosen. Which one to choose depends on the risk in case of failure and the criticality.

Formal documentation and the availability of tools
If specification or model information is available in a formal notation, this can be fed directly into test design tools, which then derive test cases. This will very much decrease the effort required to design the tests.

Conformance to standards
Industry and regulatory standards may require use of certain test techniques and coverage criteria, especially for safety critical software or software with a high integrity level.

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

Customer wishes
The customer may require specific test techniques to be used and test coverage to be achieved (when using →white box test design techniques). This is a good idea, as it generally leads to at least a minimum thoroughness of supplier testing, which may lead to less faults to be detected in customer or acceptance testing.

Risk analysis
The expectation of risk dictates more or less thorough testing, i.e. the choice of techniques and the intensity of the execution. Risk prone areas should be tested more thoroughly.

Further 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 should never be chosen by default. Their selection should always be based on a thoughtful decision. The following list should help to choose the most useful test technique.

Testing functionality

Correct functioning of the system is certainly of great relevance. A sufficient verification of the functionality of the test object has to be guaranteed in any case. Developing all test cases, regardless by which technique or procedure, includes determination of the expected results and reactions of the test object. This ensures a verification of the functionality for every executed test case. It may be distinguished if a failure exists or the correct functioning has been implemented.

Equivalence class partition combined with boundary value analysis

Equivalence class partitioning in combination with boundary value analysis should be applied for every test object to determine the test cases. When executing these test cases the according tools for measuring code coverage should be used in order to find the already achieved test coverage (see chapter 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 according behavior of the functions in an adequate way.

If dependencies between the input data are given, which must be considered 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.

Testing a whole system 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 these black box techniques. The parts of the test object still not executed should then be specifically considered for a white box test. Depending on the criticality and nature of the test object, an accordingly expensive white box technique has to be selected.

Minimum criterion: branch coverage

As minimum criterion, branch coverage should be used. If complex conditions exist in the test object then modified branch condition determination testing is the appropriate technique.
While measuring coverage, loops should be repeated more than once. At critical parts of the system, verification of the loops must be done using the according methods (*boundary interior*-path test and structured path test [Howden 75]).
Path coverage has to be seen as a mere theoretical measure and is of little importance in practice because of the great cost and because it is impossible to achieve for programs with loops.
It is reasonable to apply white box techniques at lower test levels while black box techniques offer an adequate solution for 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. It is reasonable to use the experience of the testers to find further faults.

Hint

Testing always contains 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.
Basis for the selection of the white box technique is the structure of the test object. If for example no complex conditions are included in the test object, the usage of modified condition decision coverage makes no sense.

6 Test Management

This chapter describes how to organize test teams, which team member 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 must be executed during the whole software product life cycle (see chapter 3). These testing related tasks should be coordinated and planned in close cooperation with development activities. The easiest solution is to let the developer test herself.

But the individual developer or development team tends to be blind for their own errors. Therefore it is much more effective to have different people develop and test and to organize testing as independent as possible from development.

Benefits of independent testing

The benefits of independent testing include:

Independent testers are unbiased and see other and different defects than developers;
An independent tester can verify (implicit) assumptions developers made during specification and implementation of the system.

Possible drawbacks of independent testing

But there can also be some drawbacks:

Due to too much isolation from the development team there might be a lack of communication;
Independent testing may become a bottleneck if not equipped with the necessary resources;
Developers may lose a sense of responsibility for quality, as "the testers will find the trouble anyway".

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[52], instead of their own, i.e. a developer tests the program of a colleague.

52. Often called "buddy testing".

2. There are testers within the development team; these testers do all test work in their team.
3. One or more dedicated testing teams exist within the project team (these teams are not responsible for development tasks). Such independent testers may belong to the business organization, user community or an IT operations group.
4. There are independent test specialists for specific testing tasks (such as performance test, usability test, security test, or compatibility test).
5. A separate organization (testing department, external testing facility (contractor), test laboratory) is responsible for testing (on specific test levels, e.g. system test).

When to choose which model

For each of these models, having testing consultants available would be advantageous. These consultants could support several projects and offer methodic assistance in areas like training, coaching, test automation etc. Which of the above-mentioned models is appropriate, depends – among other things – on the actual test level.

Component testing: Testing should be performed in close conjunction with the development activities. Although often applied, it is definitely the worst choice to let developers test their own programs. Independent testing organized like model 1 would certainly improve testing quality. Testing like model 2 is useful, if a sufficient number of testing people relative to the number of development staff can be made available for testing. 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 sets testing standards and rules, prepares testing schedules and requires test logs from the developers.
To provide method support, 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 analogous to component testing (models 1, 2).

If components originating from several teams are integrated, then a mixed integration team with representatives from the involved development groups or an independent integration team should be responsible. The individual development team will have its own view about the own component and therefore overlook faults. Depending on the

size of the development project and the number of components, models 3 to 5 should be considered here.

System testing: The final product shall be considered from the customer and end user point of view. Therefore, independence from the development is crucial. This leaves only models 3, 4 and 5 as professionally acceptable choices.

Example: VSR testing organization

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

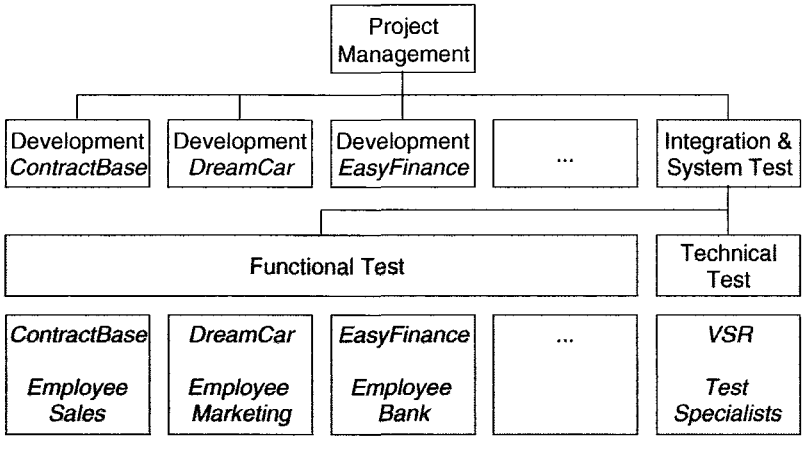


Fig. 6–1
VSR project organization

Two or three employees from each responsible user department (sales, marketing etc.) will be 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 are aware of 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 functional test cases and to perform these tests. Upon starting the testing activities, they will receive 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, non-functional tests, test automation and the support of test tools ("technical test").

A test manager, responsible for test planning and test control, will be in charge of the test team. Her tasks also comprise coaching of the test personnel, especially instruction of the staff on the subject of testing the business requirements.

6.1.2 Tasks and Qualifications

Specialists should be available whose know-how covers the full scope of activities in the test process. 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, possessing know-how and experience in the fields of software testing, quality management, project management and personnel management. Typical tasks may include:

- Writing or reviewing the test policy for the organization.
- Writing the test strategy and test plan as described in chapter 6.2.2
- Representing the testing perspective in the project.
- Procuring testing resources
- Selecting suitable test strategies and methods, Introducing or improving testing related processes (problem management, suitable configuration management) in order to be able to trace back changes and reproduce all tests;
- Initiating and monitoring the test work, i.e. the specification, implementation, and execution of tests at all test levels.
- Introducing suitable metrics for measuring test progress and evaluating the quality of the testing and the product.
- Selecting and introducing suitable test tools, and organizing required training in tool use for testers. Deciding about the kind and extension of the test environment and test automation.
- Planning the test runs and regular adaptation of the test plans based on test results and project and test progress.
- Writing →test reports and communicating them.

Test designer (test analyst): Expert(s) in test methods and test specification, having know-how and experience in the fields of software testing, software engineering and (formal) specification methods. Typical tasks may include:

- Analyzing, reviewing and assessing user requirements, specifications, designs 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 excellent knowledge of the testing tools and script languages. Automates tests as required making use of the test tools available for the project, including scripting languages.

Test administrator: Expert(s) for installing and operating the test environment (system administrator know-how). Sets up and supports the test environment (often coordinating with system administration and network management).
Tester[53]: Expert(s) in executing tests and incident reporting (IT basics, testing basics, applying the test tools, understanding of the test object). Typical tester tasks may include:
Reviewing test plans and test cases.
Using test tools and test monitoring tools (for example for performance measurement).
Executing and logging tests, evaluating the results and documenting the results and the deviations.

Certified Tester

In this context, what does the Certified Tester training offer? The basic 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 procedures are necessary. Under the supervision of a test manager, a Certified Tester can manually perform tests and document them. She is familiar with basic techniques from the field of test specification and test management. Thus also every software developer should know these basics of software testing in order to be able to adequately execute the testing tasks required by organizational models 1 and 2.

Before being able to fulfill the role of a test designer or test manager, appropriate experience as a tester should be gathered. Then the second educational level (Advanced Level) offers training for these tasks.

Even social competence is important

To be successful, in addition to technical and test specific skills, a tester also needs social skills: Ability to work in a team, political and diplomatic aptness
Skepticism: willingness to question apparent facts
Persistence and poise
Accuracy and creativity
Ability to get acquainted with (complex fields of) application fast

Multi-disciplinary team

Especially when performing system tests, it is often necessary to extend the test team by additional IT specialists, at least temporarily for performing work for the test team. For example, these might be data base administrators, data base designers or network specialists. Professional specialists from the application field of the software system currently being tested are often indispensable. Managing such a multi-disciplinary test team is a challenge even for experienced test managers.

53. The term "tester" often is also used as generic term for all above-mentioned roles.

Specialized software test service providers

If not enough appropriate resources are available within the company, test activities can also be assigned to external service providers, specializing in software testing or in certain test areas (such as performance, security or usability testing). This is similar to placing a contract for software development with an external software house. Based on their experience and their use of pre-defined solutions and procedures, these test specialists are able to rapidly deploy the optimal test for the project at hand. They can also make available any missing specialist skills from each of the above mentioned qualification profiles, for application in the project.

6.2 Test 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 planning for the quality assurance measures is needed which is documented in the quality assurance plan.

6.2.1 Quality Assurance Plan

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

Quality Assurance Plan according to IEEE 730:

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. SQAP Change Procedure and History

During this 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 starts as early as possible in the software project and is influenced by the test policy of the organizations, the scope of testing, objectives, risks, and constraints, and product criticality.

Test planning activities

The test manager's planning activities may include:

Defining the overall approach to and strategy for testing (see chapter 6.4);
Deciding about the test environment;
Definition of the test levels as well as their cooperation and integrating and coordinating 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;
Determine how much test documentation shall be prepared and deciding about templates;
Writing the test plan and deciding about what, who, when and how much testing;

Estimating test effort and test costs ; (Re-)estimating and (re-)planning the testing tasks.

The results are documented in the test plan[54]. Standard [IEEE 829] provides a reference structure:

Test Plan according to IEEE 829

1. Test plan identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach

54. "Test plan" must not be confused with the "test schedule" which means the "detailed time planning".

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[55] works well in practice. A detailed description of the listed points can be found in appendix A. The sections listed will be found in many projects in this or slightly modified form. For example separate test plans for system testing or other specific test levels are possible.

Test planning is a continuous activity for the test manager throughout all phases of the development project. The test strategy and related plans must be updated regularly, considering feedback from test activities and recognizing changing risks.

6.2.3 Prioritizing Tests

Even with good planning and control it happens that time and budget in the total test or in a certain test level are not sufficient to execute all planned test cases. In this case it is necessary to select test cases in a sensible way. It must be assured that, even with a reduced test, still as many as possible critical faults are found. This means test cases must be prioritized.

Prioritization rule

Test case prioritization should happen in such a way that a premature end of testing still assures the best possible test result at the actual point of time.

The most important test cases first

A prioritization also has the advantage that the most important test cases are executed first. This way important problems can be found early.

Below, criteria for prioritization and thus for determining the order of execution of the test cases are outlined. It depends on the

55. The current draft of the new IEEE 829 standard (2005, [IEEE 829]) 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.

project, the application area and the customer requirements which criteria are used.

Criteria for prioritization of test cases may be: *Criteria for prioritization*

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.

Risk of failure: Risk is the combination (mathematic product) of severity and failure probability. The severity is the expected damage. Such risks may for example be that the business of the customer using the software is impacted, thus leading to financial losses for the customer. Tests which may find failures with a large risk get higher priority than tests which may find failures with lower risks (see also chapter 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 then loose confidence in the other 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 do without some of the functionality, if it does not work. For other parts, this is not 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 the developer of the system architecture. Components which, when they fail, lead to severe consequences, for example a crash of the system, should be tested more intensively.

The **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 especially many faults, because development was not done with the necessary care. Prioritization in this area should

therefore be done using experience data from earlier projects run within the own organization.
Failures with a high **project risk** should be found early. These are failures requiring considerable correction work, binding resources and leading to considerable delays of the project (see chapter 6.4.3).

The project manager should define 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 deciding which test cases must be run or can be left out if there occur resource problems.

Where there are many defects, there are probably more

The following phenomenon often occurs in projects: Where many faults have been found before, more are present. In order to react on such circumstances, it must be possible to change test case priority. In the next test cycle (see chapter 6.3), additional test cases should be executed for such defect-prone test objects.

Without prioritizing test cases it is not possible to find an adequate allocation of limited test resources. The concentration of resources to high priority test cases is a must.

6.2.4 Test Exit Criteria

An important part of test planning is the definition of test exit criteria. The purpose of test exit criteria is to define when testing can be stopped (totally or within a test level). As test execution mostly is at the end of a project, time pressure and resource shortage can easily lead to random premature decisions about the end of testing. Deciding clear exit criteria during test planning helps mitigating this risk. Typical test exit criteria are:

Test coverage: how many test cases have been run (successfully), how many requirements are covered, how much code is covered ?
Product quality: number of faults found, criticality of failures, failure rates, reliability etc.
Residual risk: not executed tests, not repaired defects, incomplete coverage of requirements or code etc.
Economic constraints: allowed cost, project risks, delivery dates and market chances.

The test manager defines the project specific test exit criteria in the test strategy. During test execution, these criteria are then regularly measured and serve as the basis for decisions by test and project management (chapter 6.3.1).

6.3 Cost and Economy Aspects

Testing can be very costly and can constitute a significant cost factor in software development. The question is: How much effort is adequate for testing a specific software product? When does the testing effort overweigh the possible benefit? In order to answer this question, one has to understand the defect costs due to lack of testing. Then, one has to weigh defect costs against testing costs.

6.3.1 Costs of Defects

If verification and testing activities are reduced or cut out completely, the consequence is a higher number of unrevealed 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 the vendor may be obliged to pay for). Examples for such costs 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, costs due to installation of new versions, which might also require instruction of employees etc. Very few people think of these costs, but they can be huge. Just the time it takes to install a new version at all the customers can be enormous.

Indirect defect costs: Costs or loss of sales for the vendor that come up because the customer is dissatisfied with the product. For example, penalties or reduction of payment for not meeting contractual requirements, increased costs for the customer hotline, service, and support, bad publicity, loss of goodwill, loss of customers, even legal costs like loss of license (for example for safety critical software) etc.

Costs for defect correction: Costs for vendor work caused by fault correction. For example time needed for failure analysis, correction, test and regression test, redistribution and reinstallation, repeated instruction of the customer and users, delay of new products due to tying up the developers to maintenance of the existing product, decreasing competitiveness etc.

Risk analysis

It is hard to determine which types of cost will actually occur, how likely this is and how expensive this will be, i.e. how high the defect

risk is for a project. This risk of course depends on the kind and size of the software product, on the type of customer and the business or application area as well as the design of the contract, judicial framework etc. It also depends on the type and the number of failures, on the number of product installations as well as the number of users. There are certainly big differences between software developed customer specific and commercial off-the-shelf products. In case of doubt, all these influencing factors must be evaluated in a project specific risk analysis.

Finding faults as early as possible lowers the costs

Independent of how high the risk of a fault actually is: It is crucial to find faults as early as possible after their creation. Defect costs grow rapidly the longer time a fault remains in the product. A fault that is created very early, e.g. an error in the requirement 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 (Requirement definition, design, and programming) may even have to be partly repeated.

A reasonable typical assumption is: With every test level, the correction costs for a fault double with respect to the previous level. Investigations on this subject can be found in [URL: NIST Report].

If the customer has already installed the software product, there is additionally the 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 Costs of Testing

The most important measure to reduce or limit the risk is to plan verification and test activities. But also the factors that influence costs[56] of such testing activities are manifold. And in practice, it is very hard to quantify them. The following list shows the most important factors that a test manager should take into account when estimating the costs of testing:

56. A detailed discussion can also be found in [Pol 98] and [Pol 02].

Maturity of the development process
- Stability of the organization
- Developer's error rate
- Frequency of changes in the software
- Time pressure from unrealistic plans
- Validity, level of detail, and consistency 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 system under test
- 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 of the software (e.g. cyclomatic number, see chapter 4.2.5)

Test infrastructure
- Availability of testing tools
- Availability of test platforms, test environment and infrastructure
- Availability of and experience with testing processes, standards and procedures

Qualification of employees
- Testing experience and know-how of the testers
- Test tool and test environment experience of the testers
- Application (test object) experience and know-how of the testers
- Collaboration tester-developer-management-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[57]

57. For e.g. medical devices there are regulating agencies like FDA [URL: FDA] requiring more extensive documentation of the testing.

Test strategy
The testing objectives (themselves driven by quality requirements) and means to achieve them, like: Number and content of test levels (component, integration, system test ...)
Selection of the test techniques (black box or white box)
Schedule of the tests (start and execution of the test work in the project or in the software life cycle)

The test manager can directly influence only few of these factors. Her perspective looks like this:

The test manager's influence

Maturity of the software development process: An item that cannot be influenced in the short run; it must be accepted like it is; influence can only be exercised in the long run, through a process improvement program.
Testability of the software: Depends strongly on the maturity of the development process. A well-structured process with the corresponding reviews leads to better-structured software that is easier to test. That is why it can only be influenced in the long run through a process improvement program.
Test infrastructure: Exists normally from before but can be improved during the course of the project, if planned for. Thus, there is some potential for saving time and cost.
Qualification of employees: Can be partly influenced in the short run by the choice of test personnel. But can be improved over time by training and coaching.
Quality requirements: Are given by the customer and other stakeholders and can be partly influenced, e.g. by priority setting.
Test approach and strategy: can be chosen freely; the only aspect the test manager can influence and control in the short term.

6.3.3 Test Effort Estimation

Before defining a schedule and assigning resources the testing effort and the amount of resources needed must be estimated. For small projects this estimation can be done in one step. For larger projects separate estimations per test level and per test cycle might be necessary.

General test effort estimation approaches

In general two approaches for estimation of test effort are possible[58]:

58. For more information and articles about test estimation see [URL: RBS].

Listing all testing tasks and 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 run test cases per hour).

The effort per testing task depends on the factors described in the above chapter on testing costs. 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 effort data of similar projects or typical values usually leads to better results.

Rule of thumb

If no data at all are at hand a commonly used rule of thumb can be helpful: testing tasks (including all test levels) in typical business application development takes about 50 % of the overall project resources.

6.4 Definition of Test Strategy

A test strategy defines the project's testing objectives and means to achieve them. The test strategy therefore determines testing effort and costs. Selecting an appropriate test strategy is one of the most important planning tasks the test manager has to decide on.

The goal is to choose a test approach that optimizes the relation between costs of testing and costs of defects.

Cost-benefit relation

The test costs should of course be less than the costs that would be caused by defects and deficiencies in the final product. But very few software development organizations possess or bother to collect data material that makes it possible 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 are involved has a high influence on the strategy. We can distinguish two typical situations:

Preventive approaches: where 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. The use of the general V-model (see figure 3-1) with emphasis on design reviews will contribute a lot to prevent defects. Early test specification and preparation as well as application of reviews and static

analysis contribute to early defect finding and thus lead to reduced defect density during test execution. Especially in safety critical software, a preventive approach may be mandatory.

Reactive approaches: where 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 has to react appropriately. One very successful strategy in such a situation is called "Exploratory Testing". This is a heuristic approach where the Tester "explores" the test object and test design, execution and evaluation occur nearly concurrently (see also chapter 5.3.).

When should testing be started?

Preventative approaches should be chosen whenever possible. The analysis of the costs shows very clearly:

> The testing process should start as early as possible in the project. It should continuously accompany all phases of the project.

Example: VSR-Test Planning

In the project VSR, test planning and test documentation started immediately after the approval of the requirements document. For each requirement, at least one test case was designed. The draft test specification created like this was subjected to a review. Customer representatives, the development and the later system test staff were involved in this review. The result was that many requirements were identified as "unclear" or "incomplete". Additionally, people found wrong or insufficient test cases.

So, simply preparing reasonable tests and discussing them with developers and stakeholders helped to find many problems, long time before the first test was actually run.

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 these data. The criteria discussed in chapter 6.3. will be quantified (at least partially) and their correlation will be modeled. Amount and intensity of testing are then chosen such that some single or several 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 might be that no data are available, mathematical modeling is too complicated or because know-how is missing.

The approaches used in practice are placed in between these extremes and use (to different degrees) 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 (against the model)
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 statistical distribution of use cases (such as operational profiles); based on these distribution data the test effort is allocated;
Risk-based testing: uses information on project and product risks and directs testing to areas of greatest risk (see chapter 6.4.3);
Process- or standard-compliant approaches: use rules, recommendations and standards[59] (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.
Checklist-based (methodical) approaches: use failure and defect lists from earlier test cycles[60], lists of potential defects or risks[61] or prioritized quality criteria and other less formal methods;
Expert-oriented: use the expertise and "gut feeling" of involved experts (for the used technology or the application domain).

The above mentioned approaches are seldom used stand-alone. Mostly the test manager uses combinations of several approaches to develop the testing strategy.

6.4.3 Testing and Risk

When looking for criteria to select and prioritize testing objectives, test methods and test cases, one of the best criteria is "risk".

*Risk = damage * probability*

Risk is defined as the product of the loss or damage due to failure and the probability (or frequency) of failure. Damage comprises any consequences or loss due to failure (see chapter 6.3.1). The probability of occurrence of a product failure depends on the way the software

59. Such patterns and standards itself include best practices and heuristics.
60. Where many faults were found, there are often more. Faults often cling together and are a symptom of more faults. Extra test cases should be run through such defect-prone areas during the next test cycles.
61. A standard method here is "Failure Mode and Effects Analysis" (FMEA) [URL: FMEA].

product is used. The software's operational profile must be considered here.

Detailed estimation of risks is therefore difficult[62].

Risk factors to be considered may arise from the project as well as from the product to be delivered.

Project risks

Project risks are the risks that threaten the project's capability to deliver the product, such as:

Supplier side risks as for example the risk that a subcontractor fails to deliver, or discussions about contract issues. Project delays or even legal actions may result from this.

Often underestimated are lack of resources (total or partial lack of personnel with the necessary skills), problems of human interaction (e.g. if testers or test results do not get adequate attention) or internal political struggling (i.e. lack of cooperation between different departments).

Technical problems are a further project risk. Wrong, incomplete or infeasible requirements may easily lead to total collapse of the whole project. If new technologies, tools, programming languages or methods are employed without sufficient experience, the expected results – getting better results faster – can easily turn into the opposite. Another technical project risk is too low quality of intermediate results (design documents, program code or test cases), if this has not been detected and corrected,

Product risks

Product risks are risks resulting from problems with the delivered product, for example:

The delivered product has inadequate functional or nonfunctional quality.

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:

Assessing (and reassessing on a regular basis) what can go wrong (risks)

Prioritizing identified risks

Implementing actions to mitigate those risks.

62. A spreadsheet-based method for estimating risks or risk classes can be found at [URL: Schaefer].

An important risk mitigation activity is testing; Testing provides information about existing problems and success or failure of problem correction. Testing decreases uncertainty about risks, helps to estimate risks and to identify new risks.

Risk based Testing

Risk based testing helps to minimize and fight product risks from the beginning 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 strategy are determined on the basis of risk:

- The test techniques to be employed
- The extent of testing
- The priority of test cases

Even other risk minimizing measures such as training for inexperienced software developers are considered as alternatives or supplements.

Risk based test prioritization

Risk based prioritization of the tests makes sure that risky product parts are tested more intensively and earlier than parts with less risk. Severe problems (causing much corrective work or serious delays) are found as early as possible this way.

Opposed to this, distributing scarce test resources equally throughout all test objects makes not much sense. This approach will test critical and uncritical product parts with the same intensity. Critical parts are then not tested adequately and test resources are wasted on uncritical parts.

6.5 Test Activity Management

Test manager tasks

Every cycle through the test process (chapter 2.2, figure 2-4) normally generates change requests or fault correction requests to the developers. If faults are corrected or changes are implemented, a new version of the software emerges, and it must be tested again. Thus, in every test level the test process is executed repeatedly or cyclically.

The test manager is responsible for initiating, supervising and controlling these test cycles. Depending on the project size, a separate test manager might be responsible for each test level.

6.5.1 Test Cycle Planning

Chapter 6.2 showed the initial test planning (test strategy and overall schedule). It should be drawn up as early as possible in the project and documented in a test plan.

Detailed planning for each test cycle

This general planning must be supplemented by detailed planning for each upcoming concrete test cycle. At regular intervals, it must then be adapted to the current project situation considering the following aspects:

Development status: Compared to the original plans, the software actually available at the beginning of a test cycle may possibly have restricted or altered functionality. This may require adaptation of test specifications or test cases.
Test results: Problems revealed by previous test cycles may necessitate a change in test priorities. Corrected faults require additional retest, which also need to be planned; additional tests may also be needed when problems cannot be completely reproduced and analyzed.
Resources: Planning the current test cycle must be consistent with the current project plan; consequences of the current personnel and holiday planning, current availability of the test environment and of special test tools etc. should be considered.

Planning test effort

Taking into consideration these items, the test manager estimates effort and time requirements for the test activities and defines in detail, what test cases should be performed at what time by which tester and in which order. The result of this detailed planning is the (regression-)test plan for the upcoming 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 strategy. Only reliable, regular and simply measurable[63] metrics should be used. The following approaches can be distinguished:

Metrics for monitoring the test process

Fault- and failure based metrics: Number of encountered faults respectively generated →incident reports (per test object) in the particular release, including problem class and status, if possible including a relation to the size of the test object (lines of code), test duration or other measures (chapter 6.6).
Test case based metrics: Number of test cases specified or planned, number of test cases still →blocked (e.g. because of a fault not eliminated), number of test cases run (successful and unsuccessful).
Test object based metrics: Coverage of code, dialogues, possible installation variants, platforms etc.

63. This is the case, when the applied test tools yield such data.

Cost based metrics: Already incurred test cost, cost of the next test cycle in relation to expected benefit (prevented failure cost or reduced project risk or product risk).

Test status report

The test manager lists the respective current measurement results in her reports. After each test cycle, she writes a test status report, specifying 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
Assessment: (Subjective) assessment of the test object with respect to its maturity, possibility for release or the current degree of trust in the test object.

A template for such a report can be found in [IEEE 829].

Test exit criteria

On the one hand, the measured data serve as a means to determine the current situation and to answer the question: "How far progressed is the test?" On the other hand, the data serve as exit criterion and for answering the question: "Can the test be finished and the product be delivered?" The quality requirements to be met (thus the product's criticality), but also the available test resources (time, personnel, test tools) determine which criteria are appropriate for determining the end of the test. These test completion criteria are also documented in the test strategy or test plan. For every test completion criterion chosen it should be possible to calculate its value from the continuously collected test metrics.

Example: Test completion criteria for the VSR-System test

The test cases in the VSR project are divided into three priority levels:

Priority	Meaning
1	Test case **must** be executed
2	Test case **should** be executed
3	Test case **may** be executed

Based on this prioritization, the following test case based completion criteria for the VSR-System test were decided upon:

All test cases with priority 1 have been executed successfully
At least 60 % of the test cases with priority 2 have been run.

Product release

If the defined test exit criteria are met, project management (receiving advice from the test manager) decides, whether the corresponding test object should be released and delivered. For component and integration testing, "delivery" means passing on 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 actual application environment.

Release does not mean "bug free". The product will sure contain some undiscovered faults. And also some known ones which were rated as "not preventing release" and were therefore not corrected. The latter faults are recorded in the →incident database and will be corrected later, in the course of software maintenance (chapter 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, equipment, and tools), in order to make up for the delay and catch up on the schedule in the remaining cycles.

If no additional resources are available, the test plan itself 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 and omit all further variants. (e.g. tests are performed on one operating system instead of several). Although these adjustments lead to omission of some interesting tests, the available resources now at least can ensure the execution of the high priority test cases.

Depending on the severity of the faults and problems found, the test duration may be extended. This happens because additional test cycles become necessary, as the corrected software must be retested after each correction cycle (chapter 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 plans. Because the change in the test plan may increase the release risk. The test manager is responsible for communicating this risk openly and clearly at any time to the people responsible for the project.

6.6 Incident Management

To ensure reliable and fast elimination of failures detected by the various test levels, a well-functioning procedure for communication and administration of those incident reports is indispensable. The incident management starts during test execution or upon test run completion by evaluating the test log.

6.6.1 Test Log

Test log analysis

After each test run, at the latest upon completion of a test cycle all test logs are evaluated. Actual results are compared to the expected results. Each significant, unexpected event occurred during testing could be an indication of a test object's 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, a faulty test automation or an erroneous test execution caused the deviation (testers, too, can make mistakes).

Documenting incidents

If the problem[64] is caused by the test object, an incident report is initiated. This is done for every unexpected behavior or observed deviation from the predicted outcome documented in the test log. Possibly, an observation may be a recurrence of an observation recorded earlier. In this case, it should be examined whether the second observation yields additional information, which may make it possible to further narrow down the search for the cause of the problem. Otherwise, to prevent incident record duplication, a second recording of the same incident should not take place.

Cause-analysis is developers task

However, the testers do not have to investigate the cause of a recorded incident. This *debugging* is the developers' task.

6.6.2 Incident Reporting

In general, a project should establish a central database, in which all incidents[65] and failures discovered during testing (and possibly also during operation) are registered and administered. Personnel involved in development as well as customers and users[66] can report incidents.

64 Should the problem be caused by the tester, creating an incident report may of course also be sensible, for example if the problem calls for further analyses In this case, the incident will be reported to the tester and not to the developers

65 The ISTQB syllabus uses the term "incident" IEEE Standard 1044 uses the term "anomaly" In industry, the term "problem" or "issue" is often used

66 To simplify the following explanations, we assume that only developers and testers communicate using the problem report repository

These reports can refer to problems in the tested (parts of) programs, as well as to errors or faults in specifications, user manuals or other documents.

Incident reporting is also referred to as problem, anomaly or failure reporting. But incident reporting sounds less like "accusation". All open problems are reported; however, not every reported incident turns out to be a developers' error.

Incident reporting is not a one-way-street, as every developer can comment on reports, for example by requesting comments or clarification from a tester or by rejecting an unjustified report. Should a developer undertake corrections on a test object, these corrections will also be documented in the incident repository. 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 repository enables the test manager and the project manager to get an up-to-date and complete picture about the number and status of problems and on the progress of corrections. For this purpose, the repository should offer appropriate reporting and analysis tools.

Hint: Use an incident database

One of the first steps when introducing a systematic test process for a project should be implementing a 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 statistic analysis of the incident reports, every report shall be derived from a report template valid for the whole project. This template and reporting structure has to be defined, for example in the test strategy.

Besides the actual problem description, the incident report typically contains further information identifying the tested software, test environment, name of the tester, defects class and prioritization, as well as other information important for reproducing and localizing the fault. Table 6-1 shows an example for an incident report template:

	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 (chapter 6.6.4)
	Severity	Classification of the severity of the problem (chapter 6.6.3)
	Priority	Classification of the priority of correction (chapter 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 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

Tab. 6-1
Incident report template

A similar, slightly less complex structure can be found in [IEEE 829]. Many attributes can be further sophisticated and split up like shown in [IEEE 1044]. For example, if the incident repository 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 her 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 of 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 respon-

sible developer will understand the problem and can identify its cause with minimal effort.

Localizing the cause of problems and repairing faults is extra work for developers who normally have enough to do from before. Thus, the tester has the task to "sell" her 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 unclear or difficult to understand.

6.6.3 Incident Classification

An important criterion when judging a reported problem is its "severity", meaning the degree of impact on the operation of the system (see [IEEE 610.12]). It of course makes a major difference whether one hundred uncorrected problems in the incident database represent system breakdowns or just cosmetic mistakes in some screen layouts. A severity classification is needed and could, for example, look like table 6-2:

Tab. 6-2
Failure severity

Class	Description
1 – FATAL	System breakdown, 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 all stakeholders, but especially of the user or future user of the test object. The above classification does not indicate how fast the particular problem should be corrected. Priority of handling the problem is a different category and should not be blended with severity! When determining priority of corrections additional requirements defined by product or project management (for example correction complexity,

risk in use), as well as requirements with respect to further test execution (blocked tests) are to be taken into account. Therefore, the question, how fast a fault should be corrected, is answered by a further attribute, "fault priority" (or rather "correction priority"). Table 6-3 presents a possible classification:

Tab. 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, 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.

Incident analysis for monitoring the test process

Analyzing the severity and priority of reported incidents allow the test manager to make statements regarding product robustness or deliverability. Apart from test status determination and clarification of questions such as: "How many faults were found?", "How many of these are corrected?", "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 the course of time. In this context, the most important question is: "Does the volume of product problems still increase or does the situation seem to improve?"

Incident analysis for improving the test process

Data from incident reports can also be used to improve the test process. E.g., a comparison of data from several test objects can demonstrate which test objects show an especially small number of faults. This could mean that certain test cases have yet not been defined nor executed or on the other hand that the program has been implemented with special care and skill.

6.6.4 Incident Status

Test management must not only make sure that incidents are collected and documented properly. Test management is also 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 in all its phases. For this purpose the incident status is used. Every incident report (see table 6-1) passes a series of prede-

fined states, covering all steps from original reporting up to successful defect resolution. Table 6-4 shows an example for a incident status scheme.

Tab. 6-4
Incident status scheme

Status (set by)	**Description**
New (Tester)	A new report was written. The person reporting has included a sensible description and classification.
Open (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".
Rejected (Test manager)	The report is deemed unjustified and rejected (no fault in the test object, request for change not taken into account).
Analysis (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.
Observation (Developer)	The incident described can neither be reconstructed nor be eliminated. The report remains outstanding until further information/insights are available.
Correction (Project manager)	Based on the analysis, the project manager decides 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.
Test (Developer)	As soon as the responsible developer has corrected the problem from his point of view, the report is set to this status. The new software version containing this correction is identified.
Closed (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".
Failed (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.

Figure 6-2 demonstrates this procedure:

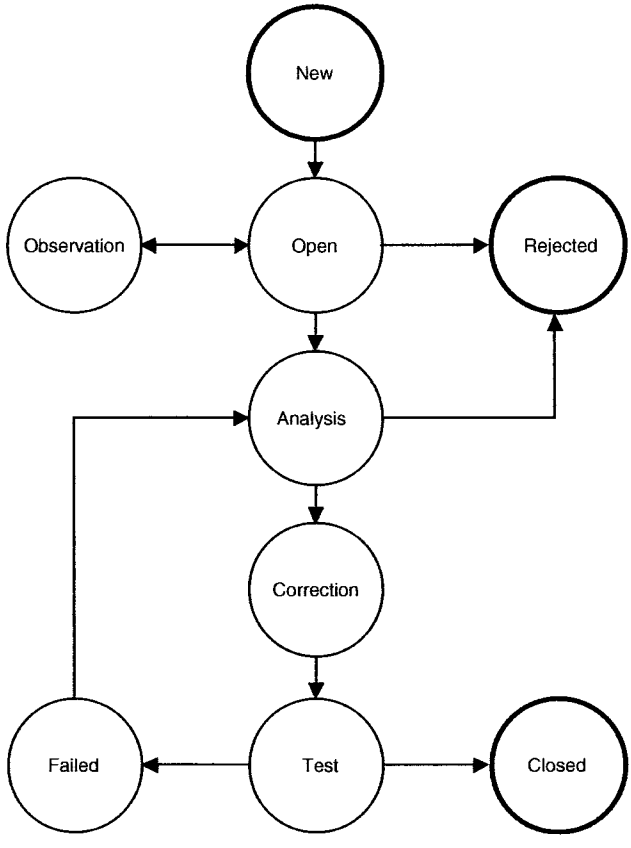


Fig. 6–2
Incident status model

A crucial fact is often ignored is, that "Closed" may only be set by the tester, and not by the developer! And this should only happen after the repeated test has proven that the problem described in the problem report does not occur anymore. Should new failures occur as side effects, after bug fixing, these failures are to be reported in new incident reports.

"Closed" may only be set by the tester

Example for extended test exit criteria for the VSR-System test

The test exit criteria for the VSR-System test shall not only reflect test progress, but also the accomplished product quality. Therefore, the test manager enhances the test exit criteria with metrics as follows:

All faults of severity "1 – FATAL" are "Closed".
All faults of severity "2 – IMMEDIATE" are "Closed".
The number of "new" incident reports per test week is stable or falling.

The scheme described above can be applied to many projects. However, the model must be tailored to cover existing or necessary project decision processes. In the above-described basic model, all decisions lie with the single person. In larger scale projects, boards make these decisions, because representatives of many stakeholders must be heard. The decision making process gets more complex.

Change control board

In many cases, changes to be performed by the developers are not real bug fixes but functional enhancements. As 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. This institution, called *change control board*, usually consists of representatives from 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, which 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. As several developers and testers take part in this process simultaneously, it is far from easy to keep an overview of the current components belonging together.

If configuration management is not done properly in a project, typical symptoms can be observed:

Typical symptoms of insufficient configuration management

- Developers mutually overwrite each other's modifications in the source code or other documents, as 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 ones
  - Because it is unclear, which versions of several components belong together and can be integrated to a larger subsystem
  - Because different versions of compiler 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, as it is unclear

Which test cases belong to which version of a test object

Which test cycle of which version of the test object gave which test results.

Testing relies on configuration management

Insufficient configuration management leads to a number of possible problems, disturbing the development and test process. If, for example, it is unclear during a →test phase whether the test objects being examined are the latest version, the tests rapidly lose their significance. A test process cannot be properly executed without reliable configuration management.

Configuration management requirements

From the prospect of the test, the following requirements should be met:

Version management: Cataloguing, filing and retrieval of different versions of a →configuration item (for example version 1.0 and 1.1 of a system). This also includes comments on the reason for the particular change.

Configuration identification: Identification and management of all files (configuration objects) in the particular version which together are forming a subsystem (configuration). Prerequisite for this is version management.

Incident status and change status accounting: Documenting of incident reports and change requests and the possibility to reconstruct their application on the configuration objects.

To check the effectiveness of configuration management, it is useful to organize **configuration audits**. Such an →audit offers the possibility to check whether all software components were documented by the configuration management, whether configurations can be correctly identified etc.

Example for configuration management in the VSR project

The software developed in the VSR project is available in different languages (for example in English, German, Chinese, 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 test always take place with consistent, valid product configurations. Similar requirements exist during system operation at the customer.

In order to implement configuration management fulfilling the above-mentioned requirements, 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

Nowadays, a multitude of standards exist setting constraints and defining the "state of the art" for software development. This is especially true for the area of software quality management and software test, as the standards quoted in this book prove.

One of the tasks for a quality or test manager is defining, in this context, which standards, rules or possibly legal directives are relevant for the product to be tested (product standards) or for the project (project standards), and to ensure these are adhered to. Possible sources are:

Company standards: Company internal directives, procedure and guidelines (also possibly set by the customer), such as the quality management handbook, a test plan template, programming guidelines.

Best practices: Not standardized, but professionally developed and widely accepted methods and procedures representing the state of the art in a particular field of application.

Quality management standards: 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 are to be performed.

Standards for particular industrial sectors: An example is standard [RTC-DO 178B] for airborne software products, defining to which minimum extent tests must be performed or documented for a particular product category or application field. Another example is [EN 50128] for railway signaling applications.

Software test standards: Process or documentation standards, defining independent of the product how software tests should be performed or documented. For example the standards [BS 7925-2], [IEEE 829], [IEEE 1028].

The standards important and relevant for software testing are covered in this book. The test concept according to IEEE 829 is described in detail in Appendix A. Following such standards also 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 and testing activities should be independently organized. The clearer the separation, the more effective testing can be performed.

Depending on the task to be executed within the test process, staff with role-specific test skills is needed. Apart from professional skills, social competence is required.

The test manager's tasks comprise the initial planning of the tests as well as further planning, monitoring and controlling the different test cycles.

In the test plan, the test manager describes and explains the test strategy (test objectives, test measures, tools etc.). The international standard [IEEE 829] provides a checklist for format and content.

Faults and deficiencies that are not found by the test and thus remain in the product can lead to very high costs. The testing strategy determines the relation between testing costs, available resources and possible defect costs.

If lack of test resources occurs, it is important to decide fast, which tests can be left out. Thus, the tests should be prioritized.

Risk is one of the best criteria for priority. Risk-based testing uses information about identified risks for planning, specification, preparation and execution of the tests. All major elements of the test strategy are determined on the basis of risk.

Measurable test exit criteria 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.

Incident reports must be collected in a project-wide standardized way and used and updated throughout 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 miscellaneous test tools supporting a tester executing his tasks. Prerequisites for the application of such tools, tool selection and implementation are discussed.

7.1 Types of Test Tools

CAST-tools

A multitude of test tools exists for supporting or automating test activities. Analogue to the term →CASE-tools (Computer Aided Software Engineering) the term →CAST-tools (Computer Aided Software Testing) is used.

There exist miscellaneous tool categories to support the different test process phases and activities. They are classified by the phases or activities they support (chapter 2.2)[66]. Inside one class of tools there are often specialized versions for specific platforms or application areas (for example performance test tools specialized for web applications).

Only in very seldom cases all available test tool categories are applied in a project. The available types of tools should however be known in order to decide when and where to apply them efficiently in a project.

Tool list

A list of commercial tools can be found at [URL: Tool-List].

The functions offered by the different tool classes are described in the following sections.

7.1.1 Tools for Test Management and Control

Test management

Test management tools offer mechanisms for easy capturing, cataloguing and administration of test cases and their priorities. They allow status tracking of the test cases, i.e. to document and evaluate if, when, how often and with which result ("passed", "failed") a test case has been executed. Some tools additionally support project management aspects during testing (i.e. resource and schedule planning for the tests).

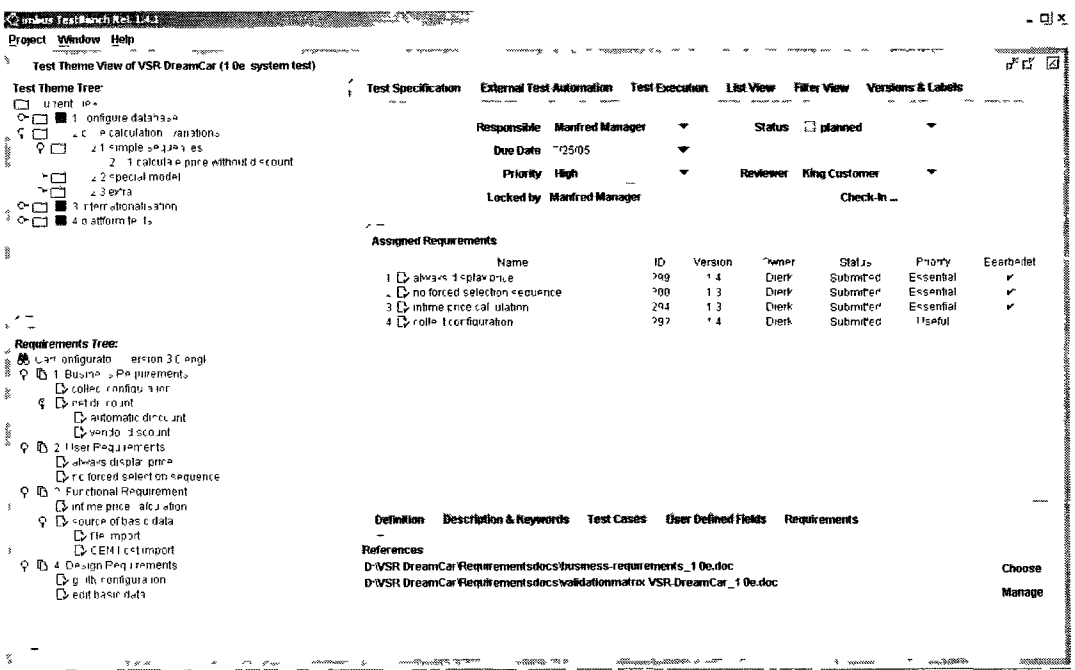
66. Especially commercially available tools often support several activities or phases and can then be classified to more than one class of tools mentioned below.

They help the test manager to plan the tests and to remain informed about the status of hundreds or thousands of test cases.

Advanced test management tools support requirements-based testing. For this purpose they allow to capture requirements (or to import them from requirements management tools) and to link them with the test cases needed for validation. Various consistency checks can be executed, for example if there is at least one test case for each requirement.

Fig. 7-1
Requirements-based test planning using TestBench [URL. TestBench]

Figure 7-1 shows, using the example of the CarConfigurator test plan, how this can look like:



Requirements management

Tools for requirements management store and administer information about requirements. They allow prioritizing requirements and tracing their implementation status.

In the narrow sense, they are not test tools, but they are of great help to derive tests from the requirements (see chapter 3.7.1) and to plan the tests relative to the implementation status of every requirement. In order to support this, requirements management tools can usually exchange data with test management tools. This allows a direct and complete connection between requirements, test cases and test results and assures a traceable validation of every requirement.

Incident management

A tool for documenting incident reports is practically indispensable to the test manager. As described in chapter 6.6, →incident management tools (also called problem tracking or defect tracking tools) are used for documentation, administration, prioritization, allocation and statistical analysis of incident reports. Advanced tools of this class include individually parameterizable incident status models. The complete workflow, from problem detection via bug fixing up to regression testing can be determined and supported. Every project team member will be guided through this workflow, according to his role in the team.

Configuration management

Configuration management tools (see chapter 6.7) also are strictly speaking no testing tools in the narrow sense. They make it possible to keep track of different versions and builds of the software. But also of the different versions of documentation and testware. Using such tools it is easier or at all possible to trace the test results of a test run on a certain test object version.

Tool integration

Integration of test tools as well as between test tools and 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 of 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 procedures. 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, i.e. a list of all test cases necessary to verify which defects have been successfully corrected in the latest test object version.

Through configuration management every code change is connected to the 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.

Generating test reports and test documentation

Both test management and incident management tools may include extensive analysis and reporting features, including the possibility to generate the complete test documentation (test plan, test specification, test report) from the contained data. The format and contents of such documents can usually be individually adjusted. Thus the

documents will be easy to 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 of a certain incident class. 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.

Test data generators

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:

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 the fact, that no expected results can be generated, (a test oracle is needed for this), and the consideration of existing code only (as with all white box methods). Faults caused by missing program instructions (code) remain undetected. Using code as a test basis for testing the code itself is in general a very poor foundation.

Interface-based test data generators analyze the test object's interface; identify the interface parameter domains and use equivalence class partitioning and boundary value analysis to derive test data from these domains. Tools are available for different kinds of interfaces, ranging from programming interfaces (Application Programming Interface, API) to Graphical User Interface (GUI) analysis. The tool is able to identify what data fields are available in a screen (e.g. numeric field, date) and generates test data, covering the respective value range (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 (see also robustness test), as specific target

values are of no importance here, but, in general, rather the fact whether the test object produces an error message or not.

Specification-based test data generators use a specification to derive test data and appropriate expected results. A precondition is, of course, that the specification is available in a formal notation. For example a method calling sequence may be given by an UML message sequence chart. The UML model is designed using a CASE-tool and is then imported by the test generator. The test generator generates test procedures, which are then passed on to a test execution tool. This approach is called model based testing (MBT).

Test designer's creativity cannot be replaced

Such test tools cannot work miracles. Specifying tests is a very challenging task, requiring a comprehensive understanding of the test object, and in addition 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. Also the corresponding expected result must be determined 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 general V-model in figure 3-1). As faults can be detected and fixed close after being introduced this decreases costs and development time.

Review support tools

Reviews are structured manual examinations using the principle that four eyes find more defects than two (see chapter 4.1). Review support tools help to plan, execute and evaluate reviews. They store information about planned and executed review meetings, participants, findings and their resolution and results. Even review aids like checklists can be included online and maintained. The collected data of many reviews can 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 prevent them.

Static analysis

Static analyzers provide measures of miscellaneous characteristics of the program code, such as the cyclomatic number and other code metrics (see chapter 4.2). Such data can be used in order to identify complex, and therefore defect-prone or risky code sections. Such areas can then be reviewed.

Static analyzers are also used for early detecting discrepancies and mistakes in the source code. These are, for example, data flow and control flow anomalies. Further examples are analyzers to enforce coding standards or link checkers for finding broken or invalid links in web site contents.

The analyzers will list all "strange" places and the list of results can grow very long. Thus, these tools are in most cases configurable, making it possible to choose the breadth and depth of analysis. When using the tool for the first time, the warning level should be chosen with a weak setting. Later on, the setting can be chosen more strong. It is very important that the setting is chosen according to project specific needs. This is decisive for the acceptance of such tools.

Model checking tools

Not only the source code can be analyzed automatically for certain characteristics. Specifications can be analyzed, too, if they are written in a formal notation or as a formal model. Analysis tools for this end are called "model checkers". They "read" the structure of a model and check their different static characteristics. For example, they can find missing states, missing transitions or other inconsistencies in the model to be checked. The specification based test generators discussed in chapter 7.1.2 are often extensions of such static "model checkers".

7.1.4 Tools for Dynamic Test

Tools take the burden of mechanical test tasks

When speaking of test tools in general, we often mean tools for automating test execution, i.e. tools for automating dynamic tests. The tester is thus released from the necessary mechanical tasks for the test execution. The tools supply the test object with test data, log the test object's reactions and record the test execution. In most cases, the tools must run on the same hardware platform as the test object itself. This, however, can have an influence on the run time behavior (like memory and processor usage) of the test object and influence the test results. This must be remembered when using such tools and evaluating test results. As such tools need to be connected to the particular test object's test interface, they vary strongly depending on the test level (component, integration, system test) they are applied in.

Debuggers

A debugger allows to execute a program or part of a program line by line, halt the execution at any line of code, and set and read program variables. Primarily, debuggers are developers' analysis tools for reproducing program failures and analyzing their causes. Additionally during testing debuggers are useful for enforcing certain special test situations, such as simulating faults, data storage overflow etc. Fault conditions like that are usually impossible to create or can only be cr

ated with disproportionately great effort. Debuggers can also serve as test interfaces during component or integration tests.

Test drivers

Test drivers or test harnesses are either commercial products or individually developed tools, offering mechanisms for executing test objects through their programming interface. Alternatively, they can be used with test objects without a user interface which are not directly accessible for a manual test. Test harnesses are mainly required during component and integration testing or for special tasks during system testing. Generic test drivers or test bed generators are also available. They perform an analysis of the programming interface of the test object and generate a test harnesses. Hence, such tools are tailored for specific programming languages or development environments. The generated test harnesses comprise the necessary initializations and calling sequences to drive the test object. If necessary, the tool also creates dummies or stubs, as well as functions for documenting target reactions and →test logging. Test harness (generators) thus significantly reduce the programming effort for the test environment. Some generic solutions (test frameworks) are available on the internet as freeware [URL: xunit].

Simulators

If performing a system test in its operational environment or using the final system is not possible or demands a disproportionally great effort, (e.g. airplane control robustness test in the airplane itself), simulators can be used. The 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 →*capture/replay or* →*capture/playback tools*, which almost completely explains their way of functioning. A test robot works in a similar way as a video recorder: The tool logs all manual inputs by the tester (keyboard inputs and mouse clicks). These inputs are then saved as a test script. This test script can be repeated automatically by "playing back". This principle sounds very tempting and easy. However, in practice, there are traps.

Excursion on the functioning of capture/playback tools

In capture mode the capture/playback tool logs keyboard inputs and mouse clicks Not only the x/y coordinates of the mouse clicks are recorded, 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, position etc) which are necessary to recognize the selected object

Capture mode

Result comparisons

In order to determine if the program under test is performing correctly, the tester can include checkpoints, i e comparison 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 an alert box, output values and texts etc)

Replay mode

The captured test scripts can be replayed and hence, in principle, be repeated as often as desired Should a discrepancy in values occur when reaching a checkpoint, "the test fails" The test robot then writes an appropriate notice in the test log file Because of their capability to perform automated comparisons of actual and expected values, test robot tools are extraordinarily well suited for regression test automation

Problem
Change of the GUI

However, one problem exists Should, in the course of program correction or program extension, the test object's GUI be changed between two test runs, it may happen that the original script does not "suit" the new GUI layout anymore Under these circumstances, the script, not being synchronized to the application any more, may come to a halt and the automated test run is aborted Test robot tools offer a certain robustness with respect to such GUI layout changes, as 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 programming

Test scripts are usually written in scripting languages These scripting languages are similar to common programming languages (BASIC-, C- or Java-like) and offer their well-known general language properties (decisions, loops, procedure calls etc) This way, 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, as capturing normally does not deliver scripts with full regression test capability The following example illustrates this

---

Example: Automated test of VSR-ContractBase

In testing the VSR-subsystem for contract documentation, it shall be examined 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;
File "Miller" contract in contract database;
Clear screen "contract data";
Read "Miller" contract back from contract database;
Compare checkpoint with screen contents;
```

When the check is successful, the contract read from the database corresponds to the contract filed before, which leads to the conclusion that the system correctly files contracts.

But when replaying this script, the tester is surprised to find the script halted unexpectedly. What happened?

Problem
Regression test capability

When the script is played a second time, when trying to file 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 try, to file the contract for the second time, by reporting:

```
"Contract already exists.
Overwrite the contract Y/N ?"
```

The test object now expects a keystroke. As this keystroke is missing in the captured test script, the automated test now halts.

Both test runs have different preconditions. As the captured script does rely on a certain precondition ("Miller" contract not in database), the test case is not regression test capable. This problem can be corrected by programming a case decision or by deleting the contract from the database as the final "cleaning-up action" of the test case.

As seen in the example, it is crucial to edit the scripts, i.e. to do programming. This requires programmer know-how. When comprehensive and long lived automation is required, a well founded test architecture must be chosen, i.e. the test scripts must be modularized.

Excursion: Test automation architectures

A good structure of the test scripts is helpful to minimize the expense for creating and maintaining automated tests A good structure also supports dividing workload 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" shall be loaded and executed, but additionally the contracts of many other customers

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, also expected results must be stored The test script leads to reading a test data line, executed the test procedure with these test data and repeats this with the next test data line If more test data are necessary, they are just added to the test data table without changing the script Even testers without programmer know-how 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 enhanced requirement is reusing test procedures For example, if contract handling should not only be tested for buying new cars, but also for buying used cars, it would be useful to run the script from the example without changes in both areas Thus, the test steps are encapsulated in a procedure with, for example, the name `check_contract(customer)` The procedure can then be called via its name and reused anywhere else

With correct granularity and corresponding well chosen test procedure names it is possible to achieve that every execution sequence available for the system user is mapped by such a procedure or command In order to make it possible to use such procedures without programmer know-how, the architecture is implemented to make the procedures callable through spreadsheet tables The (business) tester will then (analogous to the data driven test) only work with commands or keywords and test data in tables Specialized test automation programmers have to implement each of 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 grow incomprehensible. Dependencies between commands and between commands and their parameters are difficult to trace. The effort to maintain the tables grows disproportionally.

Interaction-method

The newest generation of test tools (for example [URL: TestBench]) therefore implements an object-oriented management of test modules in a database. Test modules (so called *interactions*) can be retrieved from the database by dragging and dropping them into new test sequences. The necessary test data (even complex data structures) are then automatically included. If any module is changed, every area using this modules is easy to find and can be selected. This considerably minimizes the test maintenance effort. Even very large repositories can then be used efficiently and without loosing overview.

Comparators

→Comparators (another tool class) are used in order to identify differences between expected and actual results. Comparators typically function with standard file and database formats. They detect differences between data files with expected and actual data. Test robots normally include integrated comparator functions, operating with terminal contents, GUI objects or screen content copies. These tools usually offer filtering mechanisms, skipping data or data fields irrelevant to the comparison. For example, this is necessary when date/time information is contained in the test object's file or screen output. As this information differs from test run to test run, the comparator would wrongly interpret this change as a difference between expected and actual outcome.

Dynamic analysis

During test execution, dynamic analysis tools acquire additional information on the internal state of the software being tested. E.g. information on allocation, usage and release of memory. Thus memory leaks, pointer allocation or pointer arithmetic problems can be detected).

Coverage analysis

Coverage analyzers provide structural test coverage values measured during test execution (see chapter 5.2). For this purpose, prior to test execution, an instrumentation component of the analysis 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 report is created. Most tools provide just simple coverage metrics, such as statement coverage and branch coverage (chapters 5.2.1 and 5.2.2).

7.1.5 Tools for Non-Functional Tests

Even for non-functional tests there is tool support, especially for load and performance tests.

Tools for load and performance test

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. In order to successfully use such tools and evaluate the test results, experience with performance tests is crucial.

Monitors

The necessary "measurement elements" are called "monitors".

Excursion

Load/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 By doing performance tests, the increase of response time correlated to increasing load (for example increasing number of users) can be measured, as well as the system's maximum capacity, when the software system 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 times 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, system-tuning measures (hardware extension, optimization of performance-critical software components) must be taken

Checking security

Tools for checking access and data security check a system for the possibility that not authorized persons can break into the system. Even virus scanners and firewalls can be seen as part of this tool category, as the protocols generated by such tools deliver hints about security deficiencies.

7.2 Selection and Introduction of Test Tools

Some elementary tools (e.g. comparators, coverage analyzers, primitive test drivers) are already available in several operating system environments (e.g. UNIX) as a standard. In these cases, the tester can get the necessary tool support using simple, available means. Naturally, the capabilities of such standard tools are limited, making it interesting to buy more advanced test tools available on the market.

As described above, special tools are commercially available for each 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 during the 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 not only automation tools for test execution should be taken into consideration as the one and only possible choice.

Automating chaos just gives faster chaos

The realm in which tool support may be advantageous strongly depends on the respective project environment and the maturity level of the development and test process. 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 manner (if at all); automating test execution is not a very good idea. A tool can never replace a non-existent 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, testing first must be organized. This means, first of all, that a systematic test process must be defined, introduced and adhered to. Next, thought can be given to the question, for which process steps tools can be used to enhance the productivity or quality of work. When introducing testing tools, it is recommended to adhere to the following order of introduction:

Order of tool introduction

1. Incident management
2. Configuration management
3. Test planning
4. Test execution
5. Test specification

Take into account learning curve

Some time is necessary to learn the new tool and to establish its use. This must be taken into account. Based on the learning curve, instead of the desired productivity increase, productivity may even decline for some time. It is therefore risky to introduce a new tool during "hot" project phases, hoping to solve bottlenecks there and then by introducing automation.

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. As with every investment, it is also important to consider the time frame in which the new test tool will start to pay back.

Make a cost-benefit analysis

Test execution automation tools offer a good possibility for estimating the amount of effort saved when comparing an automated test run to a manually run test. The extra test programming effort must of course be taken into account, resulting in a negative cost-benefit balance after only one automated test run. Only after further automated regression test runs have been performed (figure 7-2), achieved savings may accumulate.

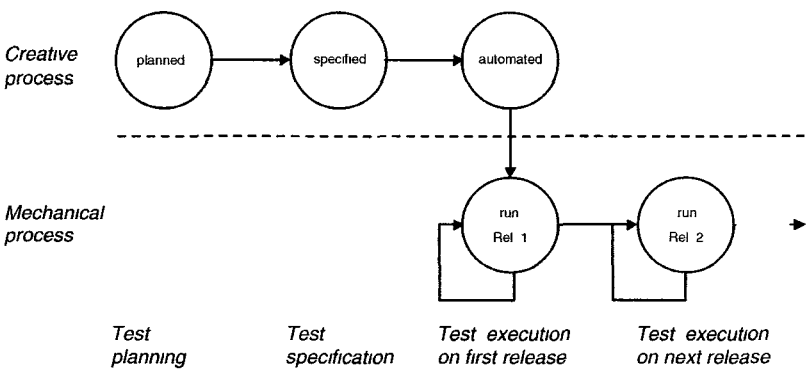


Fig. 7-2
Test case life cycle

After a certain number of regression test runs, the balance will turn out positive. It is difficult to give an exact estimate of the time for pay back. The break-even point will only be reached, 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 favorable balance is definitely possible from the third test cycle onwards for test robot tools (see [URL: imbus 98]). Of course, this kind of calculation only makes sense, when manual execution is possible at all. However, there are many tests, which cannot be run in a purely manual way (e.g performance tests). The *have to* be run automatically.

Evaluate the influence on test quality

Merely discussing test effort does not suffice. The extent of test quality improvement by applying the new test tool, resulting in detection and elimination of more faults, or resulting in more trustworthiness of the test, must also be taken into account. Development-, support- and maintenance-expenses will decrease because of this, however not before the medium-term, though the savings potential is significantly higher and therefore more interesting then. To summarize, we observe that:

The creative test activities can be supported by tools. This helps the tester improve test quality.
The mechanical test execution can be automated, reducing test effort or allowing for more tests with the same effort. More tests do not necessarily mean better tests, though.

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

The actual selection (evaluation) of the tool starts as soon as it has been clarified which test task a tool shall support. As explained earlier, the investment can become very large. It is therefore advisable to proceed carefully and in a well-planned way. The selection process consist of the following five steps:

1. Requirement specification for the tool application
2. Market research (creating an overview of possible candidates)
3. Tool demonstrations and creating a short list
4. Evaluating the tools on the short list
5. Review results and selection of the tool

For the first step, requirement specification, the following criteria may be relevant:

Selection criteria

Quality of interaction with the potential test objects
Tester know-how regarding the tool or method
Ease of integration into the existing development environment
Ease of integration with other testing tools or tools of the same supplier
Platform, on which the tool will be deployed
Manufacturer's service, reliability and market position
License conditions, price, maintenance costs

These and possible further individual criteria are compiled into a list and weighted according to their relative importance. Absolutely indispensable criteria are identified and marked as knock-out criteria[67].

Market research and short-listing

Parallel to creating a catalogue of criteria, market research takes place: a list is created, listing the available products of the interesting tool category. Product information is requested from suppliers or collected on the internet. Based on these materials, the suppliers of the

67. An example of such a criteria catalogue can be downloaded under [URL: imbus-downloads].

best candidates are invited to demonstrate their respective tools. At least some impression of the company at hand 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:

Does the tool work with the test objects and the development environment?
Are the features and quality characteristics, because of which the respective tool was considered for the final evaluation, fulfilled in reality? Advertising can promise a lot.
Can the supplier's support (also after purchase[68]) supply qualified information and help with non-standard questions?

7.2.3 Tool Introduction

After a selection has been made, the tool shall be introduced in the company. Usually, for this purpose, a pilot project is run, to verify whether the expected benefits can actually be achieved in the frame of a real project environment. The pilot project should not be run by the same people who have been involved in the evaluation, to prevent a possible conflict of interest when interpreting the evaluation results.

Pilot operation

Pilot operation shall deliver additional knowledge of the technical details of the tool, as well as experiences with the practical use of the tool and experiences about the usage environment. It shall become apparent, whether and to what extent there exists a need for training and where, if necessary, changes should be made to the test process. Furthermore, rules and conventions for the broad use should be developed, such as naming conventions for files and test cases, rules for modularizing the tests etc. If test drivers or test robots are introduced, it can be determined during the pilot project, if it is reasonable to build test libraries in order to facilitate reuse of certain tests and test modules outside the project.

Because the new tool will generate additional workload in the beginning, tool introduction requires strong and ongoing commitment of tool users and stakeholders. Thus it would be wrong to proceed with a company-wide introduction with excessive promises. Coaching and training will help to motivate the future tool users.

68 Many suppliers just refer to the general hot line after purchase

Success factors

Important success factors during roll-out are:

stepwise introduction;
integrate the tool support within 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.

That means 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 accompanying coaching

This chapter pointed out many difficulties and the additional effort when selecting and introducing tools for supporting the test process. But it 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. A careful tool introduction, however, is necessary; otherwise the wrong tool quickly becomes "shelfware" lying unused on the bookshelf.

7.3 Summary

Tools are available for every phase of the test process, helping the tester to automate test activities or improve the quality of these activities.
Use of a test tool is only beneficial, when the test process is defined and controlled.
Test tool selection must be a careful and well-managed process, as 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 regular application of the tool.

Appendix

A Test Plan according to IEEE Std. 829

This appendix describes the contents of a test plan according to IEEE Standard 829. It can be used as a guide to prepare a test plan[69].

1. Test Plan Identifier

Specify uniquely the name and version of the test plan. The identifier must make it possible to refer to this document from other project documents in a clear and precise way. A standard for document identification is often given by rules set by the project 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 components to be used are the name of the test plan, its version and its status.

2. Introduction

The introduction should give a short summary of the project background. Its intent is to help the people involved in the project (customer, management, developer, and tester) to better understand the contents of the test plan.

Part of this chapter should be a list of used documents. These include typically:

Policies and standards, such as industry standards, company standards, project standards, customer standards, the project authorization (possibly the contract), project plan and other plans, the specification.

In multilevel test plans, each lower level test plan must reference the next higher-level test plan.

3. Test Objects or Items

This section should contain a short overview of the parts and components of the product to be tested. Identify the test items including their version/revision level. Also specify characteristics of their transmittal

69. The standard is going to change in 2006 and probably a master test plan and a level test plan will replace this outline.

media and their specification. In order to avoid misunderstanding, there should be a list of what is not subject to testing.

4. Features to be tested

This section should identify all functions or characteristics of the system, which should be tested. The test specification and more detailed descriptions as well as an assignment to test levels or phases should be referenced.

5. Features not to be tested

In order to avoid misunderstanding and prevent unrealistic expectations, it should be defined which aspects of the product shall or can not be tested. (This may be due to resource constraints or technical reasons). There may also be different levels of testing for different features.

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 her status reports.

6. Test Approach or Strategy

Describe the test objectives, if possible based on risk analysis. The analysis shows which risks are imminent 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, choose and describe the test methods to be used. 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.

7. Acceptance 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[70] and delivered[71]. Acceptance- or test exit criteria are defined to achieve this.

70. Release: A management decision in which the test object is decided to be "ready".
71. Delivery may also mean: transfer to the next test level.

The criterion "defect free" is, in this context, a rather less useful criterion, because testing cannot show that a product has no faults. Normally, criteria therefore include a combination of "number of tests executed", "number of faults found " and "severity of problems found".

For example: At least 90 % of the planned tests are executed correctly and no class 1 problems (crashes) have been found.

Such acceptance criteria can be different between the test objects. The thoroughness of the criteria should be oriented depending on the risk analysis, i.e. for uncritical test objects, acceptance criteria can be weaker than for e.g. safety critical test objects. Thus, the test resources are concentrated on important system parts.

8. Suspension Criteria and Resumption Requirements

Aside from acceptance criteria, there is also a need for criteria to indicate a suspension or termination of the tests.

It may be that a test object is in such a bad shape that it has no chance to be accepted, even after an enormous amount of testing. In order to avoid such wasteful testing we need criteria which will lead to termination of such useless testing at an early enough stage. The test object will then be returned to the developer without the need to execute all tests.

Analogous to this there is the need for criteria for resumption or continuation of the tests. The responsible testers will typically execute an entry test. After this is executed without trouble, the real test begins.

Hint

Criteria should only involve measurements, which can be measured regularly, easy and reliably, for example because they are automatically collected by the used test tools. The test manager should then list and interpret these data in every test report.

---

9. Test Deliverables

In this section we describe which data and results every test activity will deliver and in which form these results are communicated. This not only means the test results in a narrow sense (for example incident reports and test protocols), but it includes also planning and preparation documents like 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 which 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. This is not in order to conceal bad news, but it should be used to assure that the right consequences are chosen after possible bad news.

10. Testing Tasks

This section is a list of all tasks necessary for the planning and execution of the tests, including assignment of responsibilities. The status of the tasks (open, in progress, delayed, done) must be followed up. This point is rather part of the normal project planning and follow up and is therefore reported in the regular project or test status reports.

11. Test Infrastructure and Environmental Needs

This section lists the elements of the test infrastructure necessary to execute the planned tests. This typically includes test platform(s), tester work places and their equipment, test tools, development environment or parts thereof necessary for the testers and other tools (e-mail, WWW, Office packages etc.).

Hint The test manager should consider the following aspects: Acquisition of the not available parts of the before 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. Such specialists may be in other departments or must be recruited from external providers.

12. Responsibilities

How is testing organized with respect to the project? Who has what authority and responsibility? Possibly the test personnel must be divided into different test groups or levels. Which people have which tasks?

Hint Responsibilities and authorities may change during the course of the project. Therefore the list of responsibilities should be a table, maybe as an appendix to the test plan.

13. Staffing and Training Needs

This section specifies the staffing needs (roles, qualifications, capacity, and when they are needed, as well as planning vacations etc). This planning is not only for the test personnel in the narrow sense, but should also include personnel for administrating the test infrastructure, developers, customers and consultants for every tool, software product (for example, database systems) and interfacing product necessary during the testing effort. Training for providing necessary skills should be included.

14. Schedule

An overall schedule for the test activities is described here, with the major milestones. This plan must be coordinated with the project plan and be 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 on test results and, if necessary, delay milestones because extra correction and testing cycles must be executed. If any test resources are shared with others, for example a test lab, this must be clearly visible in the schedule.

Hint

The test manager must assure that the test activities are included in the project plan. They must not be an independent "state in the state".

15. Risks and Contingencies

In the section about test strategy, risks in the test object or its use are addressed. This section, however, addresses risks within the testing project itself, i.e. risks when implementing the test concept, and risks resulting from not implementing reasonable activities, because there are no resources for them in the concrete project. The minimum should be a list of risks, which will then be monitored at certain points of time, in order to find measures to minimize them.

Hint

These risks should definitely be addressed:

Delays in development
Too low quality in the system under test
Problems with the test infrastructure
Lack of qualified or other key personnel

16. Approvals

This section should contain a list of people or organizations that approve the test plan, review it or at least should be informed about it. Signatures should document approval. Information to other parties should also be documented after major changes, especially changes of test strategy or changes of key personnel.

Hint Relevant persons or organizations are typically development group(s), project management, project steering committee, user and operator of the software system, customer or client, and, naturally, the testing group(s).

Depending on the project situation, the intention of the approval documented here can vary.

The ideal situation is: "You approve that the here mentioned resources will be financed and used, in order to test this system appropriately as described here."

However, the most occurring practical situation is: "Because of the lack of resources, tests can only be done in an inappropriate / minimal way. Only the most important tests are executed. You approve this way of testing and accept that release decisions based on this test bear a high risk."

17. Glossary

Testing has no tradition for using standardized terminology. Thus, the test plan should contain an explanation of the testing terms used. There is a high danger that different people will have different interpretations of test terminology otherwise. For example, just ask several people for the definition of the term "load testing".

B Important information on the curriculum and on the Certified Tester exam

The "Certified Tester Foundation Level" curriculum forms the basis of this textbook, in accordance with the ISTQB 2005 curriculum.

The respective national boards create and maintain additional national versions of the curriculum. 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 curriculum 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 under [URL: ISTQB]. The ISTQB web page 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 curriculum. The size of the individual chapters does not indicate the relevance of the presented contents for the exam. Some subjects are covered in more detail in the book. Some passages, marked as excursion, go beyond the scope of the curriculum. In any case, the exams are based on the official curricula.

The exercises and questions contained in this book should be regarded solely as practicing material and examples. They are not representative for the official examination questions.

C Exercises

Exercises to 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 random sampling.
2.5 List the main characteristics of software quality according to ISO 9126.
2.6 Define the term system reliability.
2.7 Explain the phases of the fundamental test process.
2.8 What is a test oracle?
2.9 Why should a developer not test her own programs?

Exercises to 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, too (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.
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 concept be defined?

Exercises to chapter 4

4.1 Describe the basic steps for running a review.
4.2 What different kinds of review 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 this?
4.8 What different kinds of data flow anomalies exist?

Exercises to 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 procedures.
5.4 Explain the equivalence class partition technique.
5.5 Define the test completeness criterion for equivalence class coverage.
5.6 Why is boundary value analysis a good supplement to equivalence class partitioning?
5.7 List further black box techniques.
5.8 Explain the term statement coverage.
5.9 What is the difference between statement and branch coverage?
5.10 What is the purpose of instrumentation?

Exercises to chapter 6

6.1 What basic models for division of responsibility for testing tasks between development and test can be distinguished?
6.2 Discuss the benefits and backdraws of independent testing.
6.3 Which roles are necessary in testing? Which qualifications are necessary?
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 purpose do test exit criteria serve?
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 Which idea is served by 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 a 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 basic different kinds of standards exist?

Exercises to 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 do exist?
7.4 What type of test data generator can also generate expected output values? Why can't other types of test data generators?
7.5 What is a test driver?
7.6 Explain the general way of working for a capture/playback tool.
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

The definition of most of the following terms are taken from in the "Standard Glossary of Terms used in Software Testing" Version 1.1 (September 2005), produced by the "Glossary Working Party" of the International Software Testing Qualifications Board. You can find the current version of the glossary here: [URL: ISTQB].

This glossary presents terms and definitions in software testing and related disciplines. The related terms are flagged by an underline.

abstract test case
See high level test case.

acceptance testing
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. [IEEE 610.12]

actual result
The behavior produced / observed when a component or system is tested under specified conditions.

ad hoc review
See informal review.

ad hoc testing
Testing carried out informally, no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and arbitrariness guides the test execution activity.
See also exploratory testing.

alpha testing
Simulated or actual operational testing by potential customers/users or an independent test team at the software developers' site, but outside the development organization.
Note: Alpha testing is employed 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 expectation 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]

See also defect, deviation, error, fault, failure, incident, problem, bug.

atomic (partial) condition
Boolean expression containing no Boolean operators.

EXAMPLE: "A < B" is an atomic condition but "A and B" is not. [BS 7925-1]

audit
1. 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:
2. The form or content of the products to be produced
3. The process by which the products shall be produced
4. How compliance to standards or guidelines shall be measured. [IEEE 1028]

back-to-back testing
Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies. [IEEE 610.12]

bespoke software
Software developed specifically for a set of users or customers. The opposite is off-the-shelf software.

beta testing
Operational testing by potential and/or existing customers/users at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user needs and fits within the business processes.

Note: Beta testing is employed as a form of external acceptance testing in order to acquire feedback from the market.

big-bang testing
A type of integration testing in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages. [IEEE 610.12]

See also integration testing.

black box test design techniques
Documented procedure to derive and select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

black box testing
Testing, either functional or non-functional, without reference to the internal structure of the component or system.

See also functional test design technique, requirements-based testing.

blocked test case
A test case that cannot be executed because the preconditions for its execution are not fulfilled.

bottom-up testing
An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested.

See also integration testing.

boundary value
An input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum and maximum value of a range.

boundary value analysis
A black box test design technique in which test cases are designed based on boundary values.

See also boundary value.

branch
A basic block that can be selected for execution based on a program construct in which one of two or more alternative program paths are available, e.g. case, if-then-else.

branch condition
See condition.

branch condition combination coverage
See multiple condition coverage.

branch condition combination testing
See multiple condition testing.

branch condition coverage
See condition coverage.

branch coverage
The percentage of branches that have been exercised by a test suite. 100 % branch coverage implies both 100 % decision coverage and 100 % statement coverage.

branch testing
A white box test design technique in which test cases are designed to execute branches.

bug
See defect.

business-process-based testing
An approach to testing in which test design is based on descriptions and/or knowledge of business processes.

capture/playback tool
A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e. replayed). These tools are often used to support automated regression testing.

capture/replay tool
See capture/playback tool.

CASE
Acronym for Computer Aided Software Engineering.

CAST
Acronym for Computer Aided Software Testing.
See also test automation.

cause-effect graph
A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

cause-effect graphing
A black box test design technique in which test cases are designed from cause-effect graphs. [BS 7925-2]

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 being performed.
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 white box testing.

comparator
See test comparator.

complete testing
See exhaustive testing.

component
A minimal software item that can be tested in isolation.

component integration testing
Testing performed to expose defects in the interfaces and in the interaction between integrated components.

component testing
The testing of individual software components. [IEEE 610.12]

concrete test case
Test case with concrete values for its data.
See low level test case and logical test case, abstract test case.

condition
A logical expression that can be evaluated as True or False, e.g. A>B.

condition coverage
The percentage of condition outcomes that have been exercised by a test suite. 100 % condition coverage requires each single condition in every decision statement to be tested as True and False.

condition determination coverage
The percentage of all single condition outcomes that independently affect a decision outcome that have been exercised by a test suite. 100 % condition determination coverage implies 100 % decision coverage.

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
The composition of a component or system as defined by the number, nature, and interconnections of its constituent parts.

configuration item
An aggregation of hardware, software or both, that is designated for configuration management and treated as a single entity in the configuration management process. [IEEE 610.12]

configuration management
A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [IEEE 610.12]

control flow
An abstract representation of all possible sequences of events (paths) in the execution of a component or system.

control flow anomaly
Statically detectable anomaly in the control flow of a test object (for example a not reachable statement).

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.

See also white box testing

control flow graph
A sequence of events (paths) in the execution through a component or system.

coverage
The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

cyclomatic complexity
The number of independent paths through a program. Cyclomatic complexity is defined as: L – N + 2P, where L = the number of links in a graph N = the number of nodes in a graph P = the number of disconnected parts of the graph (e.g. a calling graph and a subroutine) [McCabe 76]

cyclomatic number
See cyclomatic complexity.

data flow
An abstract representation of the sequence and possible changes of the state of data objects, where the state of an object is any of: creation, usage, or destruction. [Beizer 90]

data flow analysis
A form of static analysis based on the definition and usage of variables.

data flow anomaly
Unintended or unexpected sequence of operations on a variable.

Note: The following data flow anomalies are being distinguished: ur-anomaly: Referencing an undefined variable, dd-anomaly: Two subsequent writings to a variable without referencing this variable in between, du-anomaly: Writing (defining) of a variable followed by undefining it without referencing this variable in between.

data flow coverage
The percentage of definition-use pairs that have been exercised by a test case suite.

data flow test
A white box test design technique in which test cases are designed to execute definition and use pairs of variables.

dead code
See unreachable code.

debugger
See debugging tool.

debugging
The process of finding, analyzing and removing the causes of failures in software.

debugging tool
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 halt a program at any program statement and to set and examine program variables.

decision
A program point at which the control flow has two or more alternative routes. A node with two or more links to separate branches.

decision condition coverage
The percentage of all condition outcomes and decision outcomes that have been exercised by a test suite. 100 % decision condition coverage implies both 100 % condition coverage and 100 % decision coverage.

decision condition testing
A white box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.

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.

decision table
A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases.

decision table testing
A black box test design techniques in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table. [van Veenendaal 04]

defect
A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

defect detection percentage (DDP)
The number of defects found by a test phase, divided by the number found by that test phase and any other means afterwards.

defect management
The process of recognizing, investigating, taking action and disposing of defects. It involves recording defects, classifying them and identifying the impact. [IEEE 1044]

defect management tool
A tool that facilitates the recording and status tracking of defects. They often have workflow-oriented facilities to track and control the allocation, correction and retesting of defects and provide reporting facilities.

See also incident management tool.

defect masking
An occurrence in which one defect prevents the detection of another. [IEEE 610.12]

deficiency
Non-fulfillment of a requirement related to an intended or specified use. [ISO 9000] Synonym: defect

development process

See iterative development model and incremental development model.

development specification
1. A document that specifies the requirements for a system or component. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards. [IEEE 610.12]
2. Development phase (of the general V-model) in which the requirements for the system to be developed are collected, specified and approved.

   See also requirement and specification.

development testing
Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers. [IEEE 610.12]

See also component testing.

deviation

1. difference between a characteristical value or a value assigned to a characteristic and a reference value.
2. Deviation of the software from its expected delivery or service. See also incident.

driver

A software component or test tool that replaces a program that takes care of the control and/or the calling of a component or system. [Pol 02]

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. [IEEE 610.12]

dynamic testing

Testing that involves the execution of the software of the component or system.

efficiency

The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated condition. [ISO 9126]

efficiency testing

The process of testing to determine the efficiency of a software product.

emulator

A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. [IEEE 610.12]

See also simulator.

equivalence class

See equivalence partition.

equivalence (class) partition

A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification.

equivalence (class) partition coverage
The percentage of equivalence partitions that have been exercised by a test suite.

equivalence (class) partitioning
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 (erroneous action)
Human action that produces an incorrect result. [IEEE 610.12]

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.

error tolerance
The ability of a system or component to continue normal operation despite the presence of erroneous inputs. [IEEE 610.12]

See also robustness.

exception handling
Behavior of a component or system in response to erroneous input, from either a human user or from another component or system, or to an internal failure.

exhaustive testing
A test approach in which the test suite comprises all combinations of input values and preconditions.

Synonym: complete testing.

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 which have not been finished.

Note: Exit criteria are used by testing to report against and to plan when to stop testing. [Gilb 96]

expected result
The behavior predicted by the specification, or another source, of a component or system under specified conditions.
See also test oracle.

exploratory testing
An informal test design technique where that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. [Bach 04]

extreme programming
Agile development process, which propagates amongst other things the test-first approach.
See also test-first programming.

failure
Actual deviation of the component or system from its expected delivery, service or result. [Fenton 91]

failure priority
Determination of how pressing 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
See defect.

fault tolerance
The capability of the software product to maintain a specified level of performance in cases of software faults (defects) or of infringement of its specified interface. [ISO 9126]
See also reliability.

field testing
See beta testing.

finite state machine
A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions. [IEEE 610.12]

finite state testing
See state transition testing.

functional requirement
A requirement that specifies a function that a system or system component must be able to perform. [IEEE 610.12]

See also functionality.

functional test design technique
Documented procedure to derive and select test cases based on an analysis of the specification of the functionality of a component or system without reference to its internal structure.

See also black box test design technique.

functional testing
Testing based on an analysis of the specification of the functionality of a component or system.

See also black box testing.

functionality
The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. Sub-characteristics of functionality are suitability, accuracy, interoperability, security, compliance. [ISO 9126]

functionality testing
The process of testing to determine the functionality of a software product.

high level test case
A test case without concrete (implementation level) values for the input data and expected results. Logical operators are used, instances of the actual values are not yet defined and/or available.

See also low level test case.

incident
Any event occurring during testing that requires investigation. [IEEE 1008]

incident management
The process of recognizing, investigating, taking action and disposing of incidents. It involves recording incidents, classifying them and identifying the impact. [IEEE 1044]

incident management tool
A tool that facilitates the recording and status tracking of incidents found during testing. They often have workflow-oriented facilities to

track and control the allocation, correction and retesting of incidents and provide reporting facilities.

See also defect management tool.

incident report

A document reporting on any event that occurs during the testing which requires investigation.

incremental development model

A development life cycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this life cycle model, each subproject follows a 'mini V-model' with its own design, coding and testing phases.

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, e.g. violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. [IEEE 610.12], [IEEE 1028]

instrumentation

The insertion of additional code into the program in order to collect information about program behavior during execution, e.g. for measuring code coverage.

intake test

A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase.

See also smoke test.

integration

The process of combining components into larger assemblies.

integration testing

Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

See also component integration testing, system integration testing.

iterative development model
A development life cycle where a project is broken into a, usually large, numbers of iterations. An interaction is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product.

load testing
A test type concerned with measuring the behavior of a component or system with increasing load, e.g. number of parallel users and/or numbers of transactions to determine what load can be handled by the component or system.

See also stress testing.

logical test case
A test case without concrete values for the inputs and outputs. In most cases, conditions or equivalence classes are specified.

See high level test case.

low level test case
A test case with concrete (implementation level) values for the input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators.

See also high level test case.

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. [ISO 9126]

maintenance
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. [IEEE 1219]

management review
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. [IEEE 610.12], [IEEE 1028]

metric
A measurement scale and the method used for measurement. [ISO 14598]

milestone
A point in time in a project at which defined (intermediate) deliverables and results should be ready.

minimal multicondition coverage
See modified condition decision coverage.

mock-up
A program in the test environment that takes the place of a stub or dummy, but that contains more functionality. This makes it possible to trigger desired results or behavior.
See also dummy.

moderator
The leader and main person responsible for an inspection or review process.

modified condition decision coverage
1. See condition determination coverage.
2. Coverage percentage defined as the number of Boolean operands values shown to independently affect the decision outcome, divided by the total number of Boolean operands (atomic part conditions), multiplied by 100.

Note: RTCA-DO 178B defines Modified Condition/Decision Coverage (MC/DC) as follows – Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken on all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome.

module testing
See component testing.

monitor
A software tool or hardware device that runs concurrently with the component or system under test and supervises, records and/or analyses the behavior of the component or system. [IEEE 610.12]

multiple condition coverage
The percentage of combinations of all single condition outcomes within one statement that have been exercised by a test suite. 100 % multiple condition coverage implies 100 % condition determination coverage.

multiple condition testing
A white box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).

negative testing
Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique, e.g. testing with invalid input values or exceptions. [Beizer 90]

non-functional requirement
A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability.
See also quality objective.

non-functional testing
Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability.

N-switch testing
A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. [Chow 78]
See also state transition testing.

off-the-shelf software
A software product that is developed for the general market, i.e. for a large number of customers, and that is delivered to many customers in identical format.

operational environment
Hardware and software products installed at users' or customers' sites where the component or system under test will be used. The software may include operating systems, database management systems, and other applications.

patch
1. A modification made directly to an object program without reassembling or recompiling from the source program.

2. A modification made to a source program as a last minute fix or afterthought.
3. Any modification to a source or object program.
4. To perform a modification as in (1), (2), or (3).
5. Unplanned release of a software product with corrected files in order to, possibly in a preliminary way, correct special (often blocking) faults.
   [IEEE 610.12]

path
A sequence of events, e.g. executable statements, of a component or system from an entry point to an exit point.

path coverage
The percentage of paths that have been exercised by a test suite.

path testing
A white box test design technique in which test cases are designed to execute paths.

peer review
A review of a software work product by colleagues of the producer of the product for the purpose of identifying defects and improvements. Examples are inspection, technical review and walkthrough.

performance
The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate. [IEEE 610.12]

See also efficiency.

performance testing
The process of testing to determine the performance of a software product.

See efficiency testing.

Point of Control (PoC)
Interface used to send inputs and stimuli to the test object.

Point of Observation (PoO)
Interface used to observe and log the reactions and outputs of the test object.

postcondition
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.

predicted outcome
See expected result.

preventive software quality assurance
Use of methods, tools and procedures contributing 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 especially 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. Contains current and complete information about all identified defects.
   See also incident management tool.

quality
1. The degree to which a component, system or process meets specified requirements and/or user/ customer or user needs or expecta tions. [IEEE 610.12]
2. The degree to which a set of inherent characteristics fulfills requirements [ISO 9000]

quality assurance
Part of quality management focused on providing confidence that quality requirements will be fulfilled. [ISO 9000]

quality attribute

1. A feature or characteristic that affects an item's quality. [IEEE 610.12]
2. A set of attributes of a software product by which its quality is described and evaluated. A software quality characteristic may be refined into multiple levels of sub-characteristics. [ISO 9126]

Quality characteristics are functionality, reliability, usability, efficiency, maintainability and portability. [ISO 9126]

quality characteristic

See quality attribute.

quality objective

Something sought, or aimed for, related to quality. [ISO 9000]

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. This technique can be used for testing non-functional attributes such as reliability and performance.

regression testing

Testing of a previously tested program following modification to ensure 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

1. A particular version of a configuration item that is made available for a specific purpose. For example, a test release or a production release. [URL: BCS CM Glossary]
2. See configuration.

reliability

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. [ISO 9126]

reliability testing

The process of testing to determine the reliability of a software product.

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 sys-

tem component to satisfy a contract, standard, specification, or other formally imposed document. [IEEE 610.12]

requirements-based testing
An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g. tests that exercise specific functions or probe non-functional attributes such as reliability or usability.

resource utilization
The capability of the software product to use appropriate amounts and types of resources, for example the amounts of main and secondary memory used by the program and the sizes of required temporary or overflow files, when the software performs its function under stated conditions. [ISO 9126]

See also efficiency.

resource utilization testing
The process of testing to determine the resource utilization of a software product.

See also efficiency testing.

result
The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports and communication messages sent out.

See also actual result, expected result.

retesting
Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.

See also regression testing.

review
An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough. [IEEE 1028]

reviewable (testable)
An indicated state of the work product or document to be reviewed or tested, it being 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
Testing oriented towards exploring and providing information about product risks.

robustness
The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions. [IEEE 610.12]

See also error tolerance, fault tolerance.

robustness testing
Testing to determine the robustness of the software product.

See also negative testing.

role
Description of specific skills in software development.

safety critical system
A system whose failure may endanger human life or lead to large losses.

safety testing
The process of testing to determine the safety of a software product.

severity
The degree of impact that a defect has on the development or operation of a component or system. [IEEE 610.12]

severity class
Classification of failures according to the impact on the user, for example degree of hindrance in using the product.

(simple) condition coverage

See condition coverage.

simulator
A device, computer program or system used during testing, which behaves or operates like a given system when provided with a set of controlled inputs. [IEEE 610.12], [RTCA-DO 178B]

See also emulator.

site acceptance testing
Acceptance testing by users/customers at their site, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes, normally including hardware as well as software.

smoke test
A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details.

Note: A daily build and smoke test is among industry best practices.

See also intake test.

software development model
Framework of software development

software item
Identifiable (partial) result of the software development process.

software quality
The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs. [ISO 9126]

See also quality.

specification
A document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and, often, the procedures for determining whether these provisions have been satisfied. [IEEE 610.12]

state diagram
A diagram that depicts the states that a system or component can assume, and shows the events or circumstances that cause and/or result from a change from one state to another. [IEEE 610.12]

state transition testing
A black box test design technique in which test cases are designed to execute valid and invalid state transitions.

See also N-switch testing.

statement (source statement)
A entity in a programming language which is typically the smallest indivisible unit of execution.

statement coverage
The percentage of all statements that have been exercised by a test suite.

static analysis
Analysis of software artifacts, e.g. requirements or code, carried out without execution of these software artifacts.

static analyzer
A tool that carries out static analysis.

static testing
Testing of a component or system at requirements or implementation level without execution of any software, e.g. reviews or static code analysis.
See also static analysis.

stress testing
Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 610.12]
See also load testing.

structural testing / structure-based techniques
See white box testing.

stub
A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component. [IEEE 610.12]

syntax testing
A black box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain.

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 verify that it meets specified requirements. [Hetzel 88]

technical review
A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. [Gilb 96], [IEEE 1028]
See also peer review.

test

1. A set of one or more test cases. [IEEE 829]
2. A set of one or more test procedures. [IEEE 829]
3. A set of one or more test cases and procedures. [IEEE 829]

test automation

The use of software to perform or support test activities, e.g. test management, test design, test execution and results checking.

test basis

All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis. [Pol 02]

test bed

See test environment and test harness.

test case

A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE 610.12]

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 (objective, inputs, test actions, expected results, and execution preconditions) for a test item. [IEEE 829]

test comparator

A test tool to perform automated test comparison.

test coverage

See coverage.

test cycle

1. Execution of the test process against a single identifiable release of the test object.
2. Execution of a series of test cases.
3. Execution of the fundamental test process for exactly one version of the test object, at which end there are orders for failure repair or change to the developer.

test data
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 driver
See driver.

test effort
The necessary resources for the test process.

test environment
An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. [IEEE 610.12]

test evaluation
Analysis of the test protocol or test log in order to determine if failures have occurred.

test evaluation report
A document produced at the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned.

test execution
The process of running a test by the component or system under test, producing actual result(s).

test harness
A test environment that comprises of stubs and drivers needed to conduct a test.

test infrastructure
The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment and procedures. [Pol 02]

test item
The individual element to be tested. There usually is one test object and many test items.
See also test object.

test level
A group of test activities that are organized 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. [Pol 02]

test log
A chronological record of relevant details about the execution of tests. [IEEE 829]

test logging
The process of recording information about tests executed into a test log.

test management
The planning, estimating, monitoring and control of test activities, typically carried out by a test manager.

test method
See test technique.

test metric
A quantitative measure of a test case, test run or test cycle including measurement instructions.

test object
The component or system to be tested.
See also test item.

test objective
A reason or purpose for designing and executing a test.

test oracle
A source to determine expected results to compare with the actual result of the software under test.
Note: An oracle may be the existing system (for a benchmark), a user manual, or an individual's specialized knowledge, but should not be the code. [Adrion 82]

test phase
A distinct set of test activities collected into a manageable phase of a project, e.g. the execution activities of a test level. [Gerrard 02]

test plan
A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their

choice, and any risks requiring contingency planning. It is a record of the test planning process. (After [IEEE 829])

test planning
The activity of establishing or updating a test plan.

test procedure specification
A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. [IEEE 829]
See also test procedure.

test process
1. The fundamental test process comprises test planning and control, test analysis and design, test implementation and execution, evaluation of test exit criteria and reporting, and test closure activities.
2. The fundamental test process comprises planning, specification, execution, recording and checking for completion and Test closure activities. [BS 7925-2]

test report
See test summary report.

test result
All documents developed during the course of a test run (mostly the test log and its evaluation).
See also result.

test robot
A tool to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions in order to automate the execution of test cases.

test run
Execution of a set of test cases on a specific version of the test object.

test scenario
See test procedure specification.

test schedule
A schedule that identifies all tasks required for a successful testing effort, a schedule of all test activities and their corresponding resource requirements.

test script
Commonly used to refer to a test procedure specification, especially an automated one.

test status report
See test summary report.

test strategy
A high-level document defining the test levels to be performed and the testing within those levels for a program (one or more projects).

test suite
A set of several test cases for a component or system under test, where the post condition of one test case 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 against exit criteria. [IEEE 829]

test technique
1. Test case design technique: method used to derive or select test cases.
2. Test execution technique: method used to perform the actual test execution, e.g. manual, capture/playback tool, etc.

testability
The capability of the software product to enable modified software to be tested. [ISO 9126]
See also maintainability.

tester
A skilled professional who is involved in the testing of a component or system.

test-first programming
Software development process where test cases are developed before the code is developed. Other names are 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 spe-

cified requirements, to demonstrate that they are fit for purpose and to detect defects.

testware
Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected outcomes, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing. [Fewster 99]

tuning
Determining what parts of a program are being executed the most, and making changes to improve its performance under certain conditions. A tool that instruments a program to obtain execution frequencies of statements is a tool with this feature.

unit testing
See component testing.

unreachable code
Code that cannot be reached and therefore is impossible to execute.

use case
A sequence of transactions in a dialogue between a user and the system with a tangible result.

use case testing
A black box test design technique in which test cases are designed to execute user scenarios.

user acceptance testing
See acceptance testing.

validation
1. Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISO 9000]
2. Validation confirms that the product, as provided, will fulfill its intended use. In other words, validation ensures that "you built the right thing." [CMMI 02]
3. Determination of the correctness of the products of software development with respect to the user or customer needs and requirements. ([BS 7925-1] with additions)

verification

1. Confirmation by examination and through the provision of objective evidence that specified requirements have been fulfilled. [ISO 9000]
2. The process of evaluating a system or component to determine whether the products of the given development phase satisfies the conditions imposed at the start of that phase. [IEEE 610.12]
3. Verification confirms that work products properly reflect the requirements specified for them. In other words, verification ensures that "you built it right". [CMMI 02]

version

1. An initial release or re-release of a computer software configuration item, associated with a complete compilation or recompilation of the computer software configuration item. [IEEE 610.12]
2. An initial release or complete re-release of a document, as opposed to a revision resulting from issuing change pages to a previous release.

V-model

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.

volume testing

Testing where the system is subjected to large volumes of data.

See also resource utilization testing.

walkthrough

A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. [Freedman 90], [IEEE 1028]

See also peer review.

white box test design technique

Documented procedure to derive and select test cases based on an analysis of the internal structure of a component or system.

white box testing

Testing based on an analysis of the internal structure of the component or system.

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.

[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.

[Boehm 73] Boehm, B. W.: "Software and Its Impact: A Quantitative Assessment", Datamation, 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® Integration, Version 1.1, CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/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, Kluver, 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 00] Spillner, A.: "From V-model to W-model – Establishing the Whole Test Process", Proceedings Conquest 2000 – Workshop

on "Testing Non-Functional 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.

[Winter 98] Winter, M.: Managing Object-Oriented Integration and Regression Testing, Proceeding of the 6'th 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 basis for

the British Computer Society ISEB certification and the earlier version of the ISTQB certification. It will be revised.

[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.

[IEEE 730] IEEE Std 730-2002, IEEE Standard for Software Quality Assurance Plans.

[IEEE 828] IEEE Std 828-1998, IEEE Standard for Software Configuration Management Plans.

[IEEE 829] IEEE Std 829-1998, IEEE Standard for Software Test Documentation (under revision, new edition probably in 2006).

[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 1012-1998: IEEE Standard for Software Verification and Validation.

[IEEE 1028] IEEE Std 1028-1996: IEEE Standard for Software Reviews.

[IEEE 1044] IEEE Std 1044-1993: IEEE Standard Classification for Software Anomalies.

[IEEE 1219] IEEE Std 1219-1998: IEEE Standard for Software Maintenance.

[IEEE/IEC 12207] IEEE/EIA Std 12207-1996: Information Technology – Software life cycle processes.

[ISO 8402] ISO 8402:1994, Quality management and quality assurance – Vocabulary.

[ISO 9000] ISO 9000:2000, Quality management systems – Fundamentals and vocabulary describes the fundamentals of a QMS and

specifies the terminology for a QMS. It was developed on the basis of previous standards: ISO 8402:1994 Quality management and quality assurance – Vocabulary, and ISO 9000-1:1994 Quality management and quality assurance standards – Part 1: Guidelines for selection and use.

[ISO 9126] ISO/IEC 9126-1:2001, Software Engineering – Product quality – Part 1: Quality model, Quality characteristics and sub-characteristics.

[ISO 9241] ISO 9241-1:1997, Ergonomic requirements for office work with visual display terminals (VDTs) – Part 1: General introduction (The standard consists of totally 17 parts [URL: ISO]).

[ISO 14598] ISO/IEC 14598-1:1996, Information Technology – Software Product Evaluation – Part 1: General Overview.

[RTCA-DO 178B] RTCA-DO Std 178B, Radio Technical Commission for Aeronautics, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc., 1992.

WWW-pages[72]

[URL: BCS] *http://www.bcs.org*
British Computer Society.

[URL: BCS CM Glossary] *http://www.bcs-cmsg.org.uk/glossary.htm*
British Computer Society Configuration Management Glossary.

[URL: FDA] *http://www.fda.gov/cdrh/*
U.S. Food and Drug Administration.

[URL: FMEA] *http://de.wikipedia.org/wiki/FMEA*
Failure Mode and Effects Analysis (Fehlermöglichkeits- und Einflussanalyse).

[URL: HTML] *http://www.w3.org/MarkUp/*
HyperText Markup Language Homepage of the World Wide Web Consortium (W3C).

[URL: imbus 98] *http://www.imbus.de/download/papers/dl_whitepapers.html*
"How to Automate Testing of Graphical User Interfaces", imbus AG.

72. The referenced URLs were checked on 12 January 2006. However, it cannot be guaranteed that the URLs remain valid.

[URL: imbus-downloads] *http://www.imbus.de/download/papers/dl_whitepapers.html*
Useful articles for downloads.

[URL: ISEB] *http://www.iseb.org.uk*
Information Systems Examinations Board (ISEB).

[URL: ISTQB] *http://www.istqb.org*
International Software Testing Qualifications Board (ISTQB).

[URL: NIST Report]
http://www.mel.nist.gov/msid/sima/sw_testing_rpt.pdf
"The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards & Technology, USA, May 2002.

[URL: RBS] *http://www.rexblackconsulting.com/Pages/Library.htm*
Homepage of RBCS (Rex Black).

[URL: Schaefer] *http://home.c2i.net/schaefer/testinglinks.html*
Homepage of Hans Schaefer.

[URL: TestBench] *http://www.imbus.de/engl/produkte/testbench.shtml*
imbus TestBench.

[URL: Tool-list] *http://www.imbus.de/engl/tool-list.shtml*
Test tool list.

[URL: UML] *http://www.uml.org*
UML-Page of the Object Management Group (OMG).

[URL: V-model XT] *http://www.kbst.bund.de/doc,-304105/Federal-Government-Co-ordinati.htm*
(or *http://www.v-modell-xt.de* in German)

[URL: XML] *http://www.w3.org/XML*
Extensible Markup Language – Homepage of the W3C.

[URL: xunit] *http://www.junit.org*
Component testing framework for Java.
See also *http://opensourcetesting.org*
Open Source Tools for Software Testing Professionals.

Further useful WWW-pages

[URL: BCS SIGIST] *http://www.testingstandards.co.uk*
The British Special Interest Group for Software Testing Standards Working Party.

[URL: IEEE] *http://standards.ieee.org*
Information about IEEE Standards.

[URL: ISO] *http://www.iso.org*
The International Organization for Standardization.

[URL: SEPT] *http://www.12207.com/index.html*
Supplying Software Engineering Standards Information to the World.

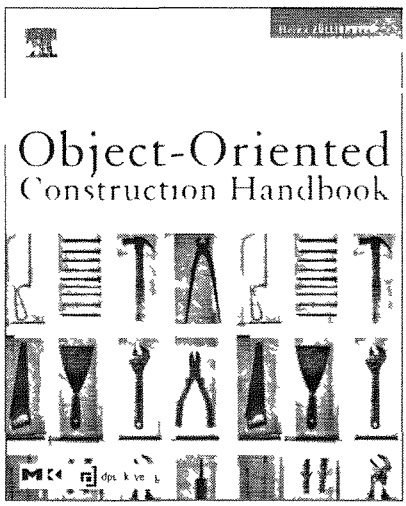
[URL: SWEBOK] *http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf*
Guide to the Software Engineering Body of Knowledge.

Index

D

E

W



2005, 544 Pages, hardback
€ 59,00 (D)
ISBN 3 89864 254 2
(Copublication with Morgan Kaufmann)

Object-Oriented Construction Handbook

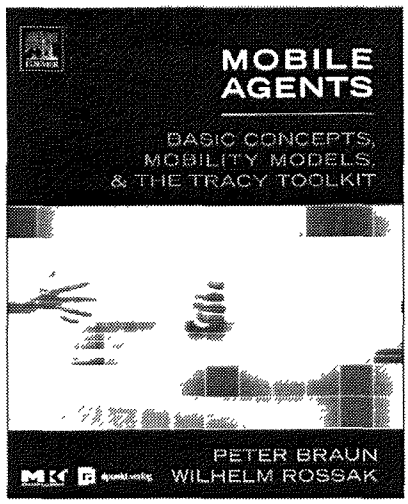
Developing Application-Oriented Software with the Tools and Materials Approach

Successful businesses and organizations are continually looking for ways to improve service and customer satisfaction in order to achieve long term customer loyalty

This book describes how the idea of customer orientation in an organization leads to the creation of appl cat on orientea software and what applicationoriented software development is and how it can be conceptually and constructively designed with object oriented techniques It describes how to best fit together the many different methodologies and techniques (such as UML Unified Process design patterns eXtreme Programming) to design and build software for real projects It br ngs together the best of research development and day to day project work to the task of building large software systems

dpunkt.verlag

Ringstraße 19 • 69115 Heidelberg
Germany
fon 0049 (0)62 21/14 83 40
fax 0049 (0)62 21/14 83 99
e mail hallo@dpunkt de
http //www dpunkt de



2005, 464 Pages, hardback
€ 49,00 (D)
ISBN 3-89864-298-4
(Copublication with Morgan-Kaufmann)

Peter Braun · Wilhelm R. Rossak

Mobile Agents

Basic Concepts, Mobility Models, and the Tracy Toolkit

Mobile agents are software nomads that act as your personal representative, working autonomously through networks. They are able to visit network nodes directly using available computing power and are not limited by platforms. This emerging field is now poised to become a cornerstone for new web-based ubiquitous computing environments.

This book provides a practical introduction to mobile agent technology and surveys the state of the art in mobile agent research. The reader can also gain hands-on experience in programming mobile agents through exploration of the source code for a complete mobile agent environment available through the companion website: www.mobile-agents.org

dpunkt.verlag

Ringstraße 19 • 69115 Heidelberg
Germany
fon 0049-(0)62 21/14 83 40
fax 0049-(0)62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de

www.dpunkt.de

Andreas Spillner • Tilo Linz • Hans Schaefer

Software Testing Foundations

Professional testing of software is becoming a more and more important task, requiring a good education. The "Certified Tester" program offers a worldwide standardized training and further education scheme for software testers.

This book provides the necessary basic knowledge and explains the concepts with a running case study. The book extensively covers the most important methods for testing software and for checking the documents produced and used during software development.

It includes:

- Fundamentals of testing
- Testing throughout the software life cycle
- Static testing techniques
- Dynamic testing techniques and test design
- Test management
- Tool support for testing

Not only testers, but programmers as well should have this basic knowledge. The book is designed for self-study. The contents comprise the necessary curriculum to pass the Certified Tester (Foundation Level) examination as defined by the International Software Testing Qualifications Board (ISTQB). It conforms to the ISTQB Foundation Level Syllabus version 2005 and also covers the latest topics such as test-first approach and risk-based testing.

From the foreword:
"This book will provide you with a solid practical foundation for your work and study of testing. Software and system testing suffers from a serious gap between best practices and common practices. If you're someone who is making a living from doing testing but haven't gotten around to reading a book, why not start with this one?"

REX BLACK,
PRESIDENT OF THE ISTQB

Topics

- **Software Test**
- **Quality Assurance**
- **Software Development**
- **Programming**

Audience

- **Software testers**
- **Programmers / software developers**
- **Test managers / project managers**
- **Students and teachers**

€ 39,00 (D)
€ 40,10 (A)

ISBN 3-89864-363-8