dpunkt.verlag

5th Edition

Andreas Spillner · Tilo Linz

Software Testing Foundations

A Study Guide for the Certified Tester Exam

- Foundation Level
- ISTQB® Compliant

About the Authors

Andreas Spillner is emeritus professor of computer science at the University of Applied Sciences Bremen. During the 1990s and early 2000s he spent 10 years as spokesman for the *TAV* (Test, Analysis, and Verification) group at the Gesellschaft für Informatik (German Computer Science Society) that he also helped to found. He is a founder member of the *German Testing Board* and was made an honorary member in 2009. He was made a fellow of the *Gesellschaft für Informatik* in 2007. His software specialty areas are technology, quality assurance, and testing.

Tilo Linz is co-founder and a board member of imbus AG, a leading software testing solution provider. He has been deeply involved in software testing and quality assurance for more than 25 years. As a founding member and chairman of the *German Testing Board* and a founding member of the *International Software Testing Qualifications Board*, he has played a major role in shaping and advancing education and training in this specialist area both nationally and internationally. Tilo is the author of *Testing in Scrum* (published by Rocky Nook), which covers testing in agile projects based on the foundations presented in this book.

Andreas Spillner · Tilo Linz

Software Testing Foundations

A Study Guide for the Certified Tester Exam

- **Foundation Level**
- **ISTQB® Compliant**

5th, revised and updated Edition

dpunkt.verlag

Andreas Spillner • *andreas.spillner@hs-bremen.de*
Tilo Linz • *tilo.linz@imbus.de*

Editor: Dr. Michael Barabas / Christa Preisendanz
Translation and Copyediting: Jeremy Cloot
Layout and Type: Josef Hegele
Production Editor: Stefanie Weidner
Cover Design: Helmut Kraus, *www.exclam.de*
Printing and Binding: mediaprint solutions GmbH, 33100 Paderborn, and Lightning Source®, Ingram Content Group.

Bibliographic information published by the Deutsche Nationalbibliothek (DNB)
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data can be found on the Internet at *http://dnb.dnb.de*.

ISBN dpunkt.verlag:
Print 978-3-86490-834-7
PDF 978-3-96910-298-5
ePUB 978-3-96910-299-2
mobi 978-3-96910-300-5

ISBN Rocky Nook:
Print 978-1-68198-853-5
PDF 978-1-68198-854-2
ePUB 978-1-68198-855-9
mobi 978-1-68198-856-6

Wieblinger Weg 17
69123 Heidelberg

Title of the German Original: Basiswissen Softwaretest
Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB®-Standard
6., überarbeitete und aktualisierte Auflage 2019
ISBN 978-3-86490-583-4

Distributed in the UK and Europe by Publishers Group UK and dpunkt.verlag GmbH.
Distributed in the U.S. and all other territories by Ingram Publisher Services and Rocky Nook, Inc.

This book is printed on acid-free paper.
Printed in Germany and in the United States.

5 4 3 2 1 0

Preface to the 5th Edition

Bestseller

The first edition of the book was published in German at the end of 2002. Since then, *Basiswissen Softwaretest* has been the best-selling book on software testing in the German-speaking world.

This 5th edition in English has been comprehensively revised and updated. It is based on the latest (6th) edition of the German-language book and the current 2018 *ISTQB® Certified Tester – Foundation Level* syllabus.

The Certified Tester training scheme

The *Certified Tester* qualification scheme is extremely successful and is widely recognized and accepted within the IT industry. It has become the de facto global standard for software testing and quality assurance education. By the end of 2020 there were over 955,000 exams taken and more than 721,000 certifications issued in 129 countries around the world [URL: ISTQB]. Many IT employment ads for beginners and experienced workers reflect this, and certified training is often an obligatory requirement. The *Certified Tester* scheme is also part of the curriculum at many universities and technical colleges.

Grass-roots knowledge required in the IT world

In spite of this rapid development, there is a lot of the grass-roots knowledge in the field of computer science that doesn't change very much over the years. We take the *Foundations* part of our book title seriously and don't discuss topics that have yet to be proven in everyday practice. Specialist topics such as web app or embedded system testing are not part of these foundations.

What's new?

This 5th edition of *Software Testing Foundations* has been comprehensively revised and extended, and its content brought completely up to date.

Side notes are not part of the official syllabus

The latest revision of the ISTQB® syllabus has seen some test techniques shifted to higher training levels, so these are no longer part of the *Foundations* syllabus. However, we have kept the corresponding sections in the book and have highlighted them as side notes. If you are using the book exclusively for exam preparation you can simply skip the side note sections.

New test techniques included

Many readers have told us that they use the book for reference in their everyday work scenarios. This is why we have included a number of additional test techniques that do not appear in the *Foundations* syllabus. These include techniques such as pair-wise testing that weren't covered in previous editions.

The case study that illustrates the implementation of the test techniques has been adapted and comprehensively updated.

We have revised the lists of standards to reflect the changes made by the introduction of ISO 29119, and all the URLs referenced in the text have been updated too.

Online resources

Any future changes to the syllabus and the glossary that affect the book text can be found on our website [URL: Softwaretest Knowledge], where you will also find exercises that relate to the individual chapters in the book. Any necessary corrections or additions to the book text are also made available at the website.

Thanks

For a book like this, success is rarely down to the authors alone, and we would like to thank all our colleagues at the German Testing Board and the *International Software Testing Qualifications Board*, without whom the *Certified Tester* program would never have achieved the global success that it enjoys. Many thanks also to Hans Schaefer, our co-author of the previous four editions of the book, for his constructive cooperation.

We would further like to thank our readers for their many comments and reviews, which have encouraged us during our work and motivated us to keep getting better. Heartfelt thanks also go to our editor Christa Preisendanz and the entire team at dpunkt.verlag for years of successful cooperation.

We wish all our readers success in the practical implementation of the testing approaches described in the book and—if you are using the book to prepare for the *Certified Tester Foundation Level* exam—we wish you every success in answering the exam questions.

Andreas Spillner and *Tilo Linz*
May 2021

Foreword by Yaron Tsubery

The software systems industry continues to grow rapidly and, especially over the last two decades, exponentially. Market requirements and a growing appetite for exciting new challenges have fuelled the development of new software technologies. These new opportunities affect almost everyone on our planet and reach us primarily via the internet and, subsequently, via smart devices and technologies.

The need for software that is easy to create and maintain has caused many key industries—such as health, automotive, defense, and finance—to open up and become visible to the world via applications and/or web interfaces. Alongside these traditional domains, new types of services (such as social media and e-commerce) have appeared and thrived on the global market. The rapid growth and enormous demands involved in introducing new software-based products that greatly impact our lifestyles and our wellbeing require new and faster ways of producing software solutions.

This situation has created a market in which multiple companies compete for market share with extremely similar products. Such competition is beneficial to consumers (i.e., software users) and, as a result, software-based products have started to become commoditized. Software manufacturers have begun to think more economically, generating increased revenues using fewer resources (i.e., doing more with less). Continual introduction of new products into our daily lives has given rise to the "agile" design and production ethos—driving a cultural change in the tradition software development life cycle, as well as pushing forward the necessity of more and early automatic tests (e.g. as driven by the DevOps movement)—that is increasingly commonplace in today's software industry, while the business leaders behind software-based products have understood that the world is becoming smaller and that competition is getting fiercer all the time. An increasingly short time to market is essential not only for generating revenue, but also simply to survive in today's market. Successful and innovative companies understand that they need to put the customer first if they want to maintain product quality,

generate brand loyalty, and increase their market share. In other words, the software industry has understood the importance of the customer to the overall product life cycle.

We in the software testing business have always known the importance of quality to the customer, because part of our job is to represent the customer's point of view. The challenges we face have grown with the complexity of software products, and we sometimes still find ourselves having to justify the necessity for software testing, even if it has become a largely standard practice within the software industry. Recently, the rise of software-based artificial intelligence (AI)—such as software enhancement in robots and autonomous devices—has created a whole new set of challenges.

Software testing is an extremely important factor in the industry. Alongside controlling costs and quality, the main issue is customer focus. Preserving a healthy balance between cost and quality is an essential customer requirement, making it critical to have well-trained and highly professional people assigned to quality and software testing roles. Recruiting skilled professionals is the key to success. The primary factors we look for when recruiting are related to a person's knowledge and skills. We look at the degree to which a person is aligned with the software testing profession, and with the required technology and industry domain (such as web, mobile, medical devices, finance, automotive, and so on). We also have to ask ourselves whether a person is suited to work in the product domain itself (for example, when candidates come from competitors). Communications and soft skills that fit in with the team/group/company are important too. In the case of industry newcomers, we have to consider how much potential a person has. This book teaches the fundamentals of software testing and provides a solid basis for enhancing your knowledge and experience through constant learning from external sources, your own personal experience, and from others.

When reading an educational book, I expect it to be sequentially structured and easy to understand. This book is based on the *Certified Tester Foundation Level (CTFL)* syllabus, which is part of the ISTQB® (International Software Testing Qualifications Board) education program. The ISTQB® has created a well-organized and systematic training program that is designed to teach and qualify software testers in a variety of roles and domains. One of the primary objectives of the ISTQB® program is to create professional and internationally accepted terminology based on knowledge and experience. The chapters in the book are designed to take you on that journey and provide you with the established and cutting-edge

fundamentals necessary to becoming a successful tester. They combine comprehensive theory with detailed practical examples and side notes that will enhance and broaden your view of software systems and how to test them. This book provides a great way to learn more about software testing for anyone who is studying the subject, thinking about joining the software testing profession, or for newcomers to the field.

For those who already have a role in software testing, the practical examples provided (based on a case study and corresponding side notes) are sure to help you learn. They provide a great basis for comparison with and application to your own real-world projects. This book contains a wealth of great ideas that will help you to build and improve your own software testing skills. The new, revised edition is based on the latest (2018) ISTQB® CTFL, which has been updated to cover agile processes and experience gained from changes that have taken place within the industry over the last few years. It also includes references to the other syllabi and professional content upon which it is based, and an updated version of the case study introduced in earlier editions. The case study is based on a multilayer solution that includes both specific and general technical aspects of software system architecture. The case study in this edition is based on a new-generation version of the system detailed in previous editions, thus enabling you to learn from a practical, project-based viewpoint.

The world is changing fast every day. Some of the technologies that we use today will become obsolete within a few years and the products we build will probably become obsolete even sooner. Software is an integral and essential part of virtually all the technology that surrounds us. Along with growth and expansion in the artificial intelligence (AI) arena and other new technologies that have yet to be introduced, this continual change offers new and exciting opportunities for the software testing profession. We are sure to find ourselves tuning our knowledge and experience in various ways, and we may even find ourselves teaching and coaching not only humans but also machines and systems that test products for us.

The fundamental knowledge, grass-roots experience, and practical examples provided by this book will prepare you for the ever-changing world and will shape your knowledge to enable you to test better and, in the future, perhaps pass on your knowledge to others.

I wish you satisfying and fruitful reading.

Yaron Tsubery
Former ISTQB® President
President ITCB®

Overview

Contents

1 Introduction

Software is everywhere! Nowadays there are virtually no devices, machines, or systems that are not partially or entirely controlled by software. Important functionality in cars—such as engine or gear control—have long been software-based, and these are now being complemented by increasingly smart software-based driver assist systems, anti-lock brake systems, parking aids, lane departure systems and, perhaps most importantly, autonomous driving systems. Software and software quality therefore not only govern how large parts of our lives function, they are also increasingly important factors in our everyday safety and wellbeing.

Equally, the smooth running of countless companies today relies largely on the reliability of the software systems that control major processes or individual activities. Software therefore determines future competitiveness. For example, the speed at which an insurance company can introduce a new product, or even just a new tariff, depends on the speed at which the corresponding IT systems can be adapted or expanded.

High dependency on reliable software

Quality has therefore become a crucial factor for the success of products and companies in the fields of both technical and commercial software.

Most companies have recognized their dependence on software, whether relying on the functionality of existing systems or the introduction of new and better ones. Companies therefore constantly invest in their own development skills and improved system quality. One way to achieve these objectives is to introduce systematic software evaluation and testing procedures. Some companies already have comprehensive and strict testing procedures in place, but many projects still suffer from a lack of basic knowledge regarding the capacity and usefulness of software testing procedures.

Grass-roots knowledge of structured evaluation and testing

This book aims to provide the basic knowledge necessary to set up structured, systematic software evaluation and testing techniques that will help you improve overall software quality.

This book does not presume previous knowledge of software quality assurance. It is designed for reference but can also be used for self-study.

The text includes a single, continuous case study that provides explanations and practical solutions for each of the topics covered.

This book is aimed at all software testers in all types of companies who want to develop a solid foundation for their work. It is also for programmers and developers who have taken over (or are about to take over) existing test scenarios, and it is also aimed at project managers who are responsible for budgeting and overall procedural improvement. Additionally, it offers support for career changers in IT-related fields and people involved in application approval, implementation, and development.

Especially in IT, lifelong learning is essential, and software testing courses are offered by a broad range of companies and individuals. Universities, too, are increasingly offering testing courses, and this book is aimed at teachers and students alike.

Certification program for software testers

The ISTQB® *Certified Tester* program is today seen as the worldwide standard for software testing and quality assurance training. The ISTQB® (*International Software Testing Qualifications Board*) [URL: ISTQB] coordinates qualification activities in individual countries and ensures the global consistency and comparability of the syllabi and exam papers. National *Testing Boards* are responsible for publishing and maintaining local content as well as the organization and supervision of exams. They also approve courses and offer accreditation for training providers. Testing Boards therefore guarantee that courses are of a consistently high standard and that participants end up with an internationally recognized certificate. Members of the Testing Boards include training providers, testing experts from industrial and consulting firms, and university lecturers. They also include representatives from trade associations.

Three-stage training scheme

The *Certified Tester* training scheme is made up of units with three levels of qualification. For more details, see the ISTQB® [URL: ISTQB] website. The basics of software testing are described in the *Foundation Level* syllabus. You can then move on to take the *Advanced Level* exam, which offers a deeper understanding of evaluation and testing skills. The *Expert Level* certificate is aimed at experienced software testing professionals, and consists of a set of modules that cover various advanced topics (see also section 6.1.2). In addition, there are syllabi for agile software development (foundation and advanced level) as well as special topics from the testing area (for example, Security Tester, Model-Based Tester, Automotive Software Tester).

This book covers the contents of the *Foundation Level* syllabus. You can use the book for self-study or in conjunction with an approved course.

Chapter overview

The topics covered in this book and the basic content of the F*oundation Certificate* course are as follows:

Software testing basics

Chapter 2 discusses the basics of software testing. Alongside the concepts of when to test, the objectives to aim for, and the required testing thoroughness, it also addresses the basic concepts of testing processes. We also talk about the psychological difficulties that can arise when you are looking for errors in your own work.

Lifecycle testing

Chapter 3 introduces common development lifecycle models (sequential, iterative, incremental, agile) and explains the role that testing plays in each. The various test types and test levels are explained, and we investigate the difference between functional and non-functional testing. We also look at regression testing.

Static testing

Static testing (i.e., tests during which the test object is not executed) are introduced in Chapter 4. Reviews and static tests are used successfully by many organizations, and we go into detail on the various approaches you can take.

Dynamic testing

Chapter 5 addresses testing in a stricter sense and discusses "black-box" and "white-box" dynamic testing techniques. Various test techniques and methods are explained in detail for both. We wrap up this chapter by looking at when it makes sense to augment common testing techniques using experience-based or intuitive testing techniques.

Test management

Chapter 6 discusses the organizational skills and tasks that you need to consider when managing test processes. We also look at the requirements for defect and configuration management, and wind up with a look at the economics of testing.

Test tools

Testing software without the use of dedicated tools is time-consuming and extremely costly. Chapter 7 introduces various types of testing tools and discusses how to choose and implement the right tools for the job you are doing.

Case Study: VirtualShowRoom VSR-II

Most of the processes described in this book are illustrated using a case study based on the following scenario:

A car manufacturer has been running an electronic sales system called *VirtualShowRoom (VSR)* for over a decade. The system runs at all the company's dealers worldwide:

- Customers can configure their own vehicle (model, color, extras, and so on) on a computer, either alone or assisted by a salesperson. The system displays the available options and immediately calculates the corresponding price. This functionality is performed by the *DreamCar* module.

- Once the customer has selected a configuration, he can then select optimal financing using the *EasyFinance* module, order the vehicle using the *JustInTime* module, and select appropriate insurance using the *NoRisk* module. The *FactBook* module manages all customer and contract data.

The manufacturer's sales and marketing department has decided to update the system and has defined the following objectives:

- *VSR* is a traditional client-server system. The new *VSR-II* system is to be web-based and needs to be accessible via a browser window on any type of device (desktop, tablet, or smartphone).
- The *DreamCar*, *EasyFinance*, *FactBook, JustInTime*, and *NoRisk* modules will be ported to the new technology base and, during the process, will be expanded to varying degrees.
- The new *ConnectedCar* module is to be integrated into the system. This module collects and manages status data for all vehicles sold, and communicates data relating to scheduled maintenance and repairs to the driver as well as to the dealership and/or service partner. It also provides the driver with various additional bookable services, such as a helpdesk and emergency services. Vehicle software can be updated and activated "over the air".
- Each of the five existing modules will be ported and developed by a dedicated team. An additional team will develop the new *ConnectedCar* module. The project employs a total of 60 developers and other specialists from internal company departments as well as a number of external software companies.
- The teams will work using the *Scrum* principles of agile development. This agile approach requires each module to be tested during each iteration. The system is to be delivered incrementally.
- In order to avoid complex repeat data comparisons between the old and new systems, *VSR-II* will only go live once it is able to duplicate the functionality provided by the original *VSR* system.

Within the scope of the project and the agile approach, most project participants will be confronted or entrusted with test tasks to varying degrees. This book provides the basic knowledge of the test techniques and processes required to perform these tasks. Figure 1-1 shows an overview of the planned *VSR-II* system.

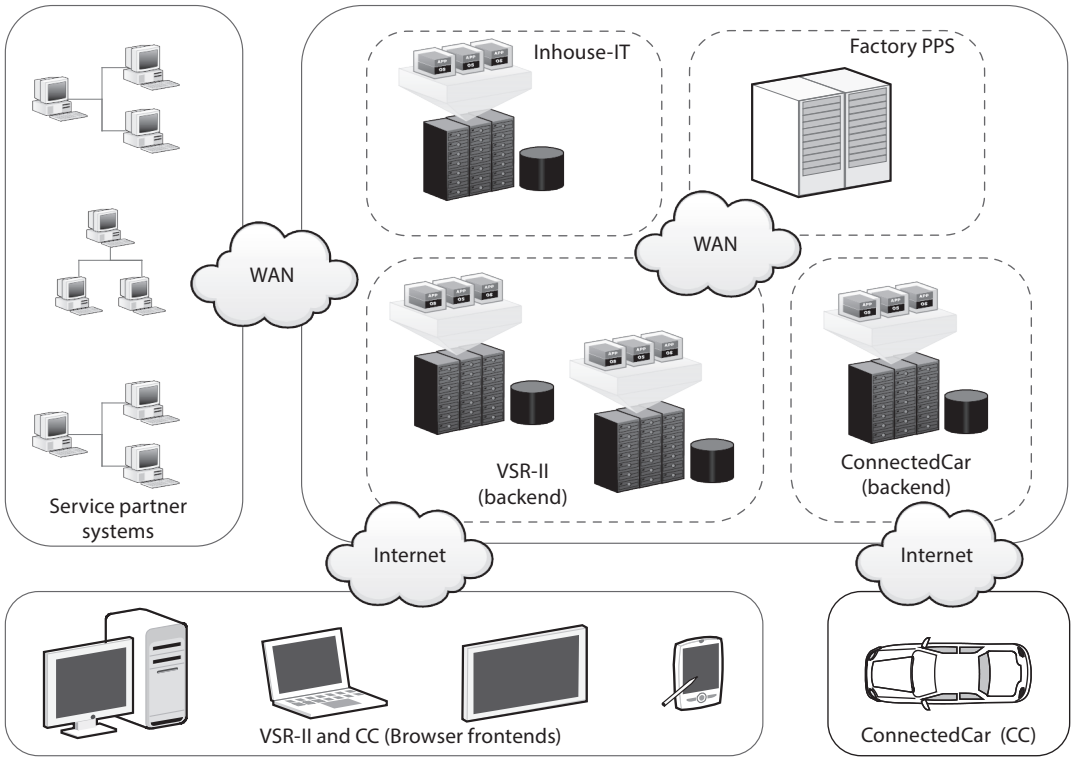

Fig. 1-1 *VSR-II overview*

The appendices at the end of the book include references to the syllabus and *Certified Tester* exam, a glossary, and a bibliography. Sections of the text that go beyond the scope of the syllabus are marked as side notes.

Certified Tester syllabus and exam

The book's website [URL: Softwaretest Knowledge] includes sample exam questions relating to each chapter, updates and addenda to the text, and references to other books by authors whose work supports the *Certified Tester* training scheme.

The book's website

We have put a free implementation of *VSR-II* as a test object online for training purposes[1]. It reproduces the *VSR-II* examples included in the book on a realistic, executable system, so you can "test" live to find the software bugs hidden in *VSR-II* by applying the test techniques presented in the book. It takes just a few mouse clicks to get started:

Web-based Training System vsr.testbench.com

1. Open your browser and load *vsr.testbench.com*
2. Create your personal *VSR-II* training workspace
3. Log into your *VSR-II* workspace and start

Fig. 1-2 *VSR-II Training System Login-Screen*

Also included in your registration for a *VSR-II* training workspace is a free basic license for the test management system *TestBench CS*, which includes the *VSR-II* test specification as a demo project and several of the *VSR-II* test cases presented in the book.

You can use *TestBench CS* not only for learning and training, but also for efficient testing of your own "real" software. A description of all features can be found at [URL: TestBench CS].

Many thanks to our colleagues at imbus Academy, imbus JumpStart and imbus TestBench CS Development Team for this awesome implementation of the VSR-II Case Study as a web-based training system.

2 Software Testing Basics

This introductory chapter explains the basic concepts of software testing that are applied in the chapters that follow. Important concepts included in the syllabus are illustrated throughout the book using our practical VSR-II case study. The seven fundamental principles of software testing are introduced, and the bulk of the chapter is dedicated to explaining the details of the testing process and the various activities it involves. To conclude, we will discuss the psychological issues involved in testing, and how to avoid or work around them.

2.1 Concepts and Motivations

Quality requirements

Industrially manufactured products are usually spot-checked to make sure they fulfill the planned requirements and perform the required task. Different products have varying quality requirements and, if the final product is flawed or faulty, the production process or the design has to be modified to remedy this.

Software is intangible

What is generally true for industrial production processes is also true for the development of software. However, checking parts of the product or the finished product can be tricky because the product itself isn't actually tangible, making "hands-on" testing impossible. Visual checks are limited and can only be performed by careful scrutiny of the development documentation.

Faulty software is a serious problem

Software that is unreliable or that simply doesn't perform the required task can be highly problematic. Bad software costs time and money and can ruin a company's reputation. It can even endanger human life—for example, when the "autopilot" software in a partially autonomous vehicle reacts erroneously or too late.

Testing helps to assess software quality

It is therefore extremely important to check the quality of a software product to minimize the risk of failures or crashes. Testing monitors software quality and reduces risk by revealing faults at the development stage. Software testing is therefore an essential but also highly complex task.

Case Study: The risks of using faulty software

Every release of the *VSR-II* system has to be suitably tested before it is delivered and rolled out. This aims to identify and remedy faults before they can do any damage. For example, if the system executes an order in a faulty way, this can cause serious financial problems for the customer, the dealer and the manufacturer, as well as damaging the manufacturer's image. Undiscovered faults like this increase the risk involved in running the software.

Testing involves taking a spot-check approach

Testing is often understood as spot-check execution[1] of the software in question (the test object) on a computer. The test object is fed with test data covering various test cases and is then executed. The evaluation that follows checks whether the test object fulfills its planned requirements.[2]

Testing involves more than just executing tests on a computer

However, testing involves much more than just performing a series of test cases. The test process involves a range of separate activities, and performing tests and checking the results are just two of these. Other testing activities include test planning, test analysis, and the design and implementation of test cases. Additional activities include writing reports on test progress and results, and risk analysis. Test activities are organized differently depending on the stage in a product's lifecycle. Test activities and documentation are often contractually regulated between the customer and the supplier, or are based on the company's own internal guidelines. Detailed descriptions of the individual activities involved in software testing are included in sections 2.3 and 6.3.

Static and dynamic testing

Alongside the dynamic tests that are performed on a computer (see Chapter 5), documents such as requirement specifications, user stories, and source code also need to be tested as early as possible in the development process. These are known as static tests (see Chapter 4). The sooner faults in the documentation are discovered and remedied, the better it is for the future development process, as you will no longer be working with flawed source material.

Verification and validation

Testing isn't just about checking that a system fulfills its requirements, user stories, or other specifications; it is also about ensuring that the product fulfills the wishes and expectations of its users in a real-world environment. In other words, checking whether it is possible to use the system as intended and making sure it fulfills its planned purpose. Testing therefore also involves validation (see Principle #7 in section 2.1.6—*"Absence-of-errors is a fallacy"*).

1. Here, we are referring to the dynamic testing processes discussed in Chapter 5. Static testing (see Chapter 4) doesn't require the software to be executed.
2. Testing alone cannot prove that all requirements have been fulfilled (see below).

No large system is fault-free

There is currently no such thing as a fault-free software system, and this situation is unlikely to change for systems above a given degree of complexity or those with a large number of lines of code. Many faults are caused by a failure to identify or test for exceptions during code development—things like failing to account for leap years, or not considering constraints when it comes to timing or resource allocation. It is therefore common—and sometimes unavoidable—that software systems go live, even though faults still occur for certain combinations of input data. However, other systems work perfectly day in day out in all manner of industries.

Freedom from faults cannot be achieved through testing

With the exception of very small programs, even if every test you perform returns zero defects, you cannot be sure that additional tests won't reveal previously undiscovered faults. It is impossible to prove complete freedom from faults by testing.

2.1.1 Defect and Fault Terminology

The test basis as a starting point for testing

A situation can only be classed as faulty if you define in advance what exactly is supposed to happen in that situation. In order to make such a definition, you need to know the requirements made of the (sub)system you are testing as well as other additional information. In this context, we talk about the test basis against which tests are performed and that serves as the basis for deciding whether a specific function is faulty.

What counts as a defect?

A defect is therefore defined as a failure to fulfill a predefined requirement, or a discrepancy between the actual behavior (at run time or while testing) and the expected behavior (as defined in the specifications, the requirements, or the user stories). In other words, when does the system's behavior fail to conform to its actual requirements?

Unlike physical systems, software systems don't fail due to age or wear. Every defect that occurs is present from the moment the software is coded, but only becomes apparent when the system is running.

Faults cause failures

System failures result from faults and only become apparent to the tester or the user during testing or at run-time. For example, when the system produces erroneous output or crashes.

We need to distinguish between the effects of a fault and its causes. A system failure is caused by a fault in the software, and the resulting condition is considered to be a defect. The word "bug" is also used to describe defects that result from coding errors, such as an incorrectly programmed or forgotten instruction in the code.

Defect masking

It is possible that a fault can be offset by one or more other faults in other parts of the program. Under these circumstances, the fault in question only becomes apparent when the others have been remedied. In other words, correcting a fault in one place can lead to unexpected side effects in others.

Not all faults cause system failures, and some failures occur never, once, or constantly for all users. Some failures occur a long way from where they are caused.

A fault is always the result of an error or a mistake made by a person—for example, due to a programming error at the development stage.

People make errors

Errors occur for various reasons. Some typical (root) causes are:

- All humans make errors!
- Time pressure is often present in software projects and is a regular source of errors.
- The complexity of the task at hand, the system architecture, the system design, or the source code.
- Misunderstandings between participants in the project—often in the form of differing interpretations of the requirements or other documents.
- Misunderstandings relating to system interaction via internal and external interfaces. Large systems often have a huge number of interfaces.
- The complexity of the technology in use, or of new technologies previously unknown to project participants that are introduced during the project.
- Project participants are not sufficiently experienced or do not have appropriate training.

A human error causes a fault in part of the code, which then causes some kind of visible system failure that, ideally, is revealed during testing (see figure 2-1: Debugging, see below). Static tests (see Chapter 4) can directly detect faults in the source code.

System failures can also be caused by environmental issues such as radiation and magnetism, or by physical pollution that causes hardware and firmware failures. We will not be addressing these types of failures here.

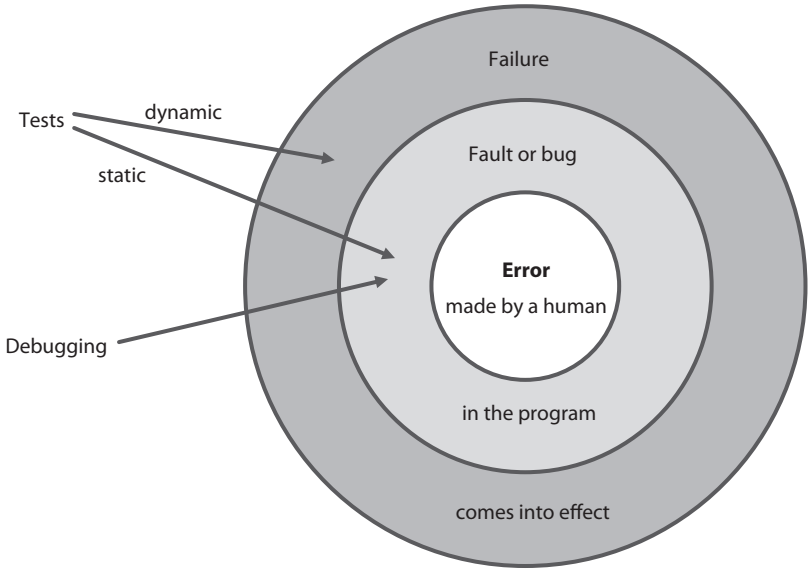

Fig. 2-1
The relationships between, errors, faults, and failures

False positive and false negative results

Not every unexpected test result equates to a failure. Often, a test will indicate a failure even though the underlying fault (or its cause) isn't part of the test object. Such a result is known as a "false positive". The opposite effect can also occur—i.e., a failure doesn't occur even though testing should reveal its presence. This type of result is known as a "false negative". You have to bear both of these situations in mind when evaluating your test results. Your result can also be a "correct positive" (failure revealed by testing) or a "correct negative" (expected behavior confirmed by testing). For more detail on these situations, see section 6.4.1.

Learning from your mistakes

If faults and the errors or mistakes that cause them are revealed by testing it is worth taking a closer look at the causes in order to learn how to avoid making the same (or similar) errors or mistakes in future. The knowledge you gain this way can help you optimize your processes and reduce or prevent the occurrence of additional faults.

Case Study: Vague requirements as a cause of software faults

Customers can use the *VSR EasyFinance* module to calculate various vehicle-financing options. The interest rate the system uses is stored in a table, although the purchase of vehicles involved in promotions and special offers can be subject to differing interest rates.

VSR-II is to include the following additional requirement:

REQ: If the customer agrees to and passes an online credit check, the *EasyFinance* module applies an interest rate from a special bonus interest rate table.

The author of this requirement unfortunately forgot to clarify that a reduction in the interest rate is not permissible for vehicles sold as part of a special offer. This resulted in this special case not being tested in the first release. In turn, this meant that customers were offered low interest rates online but were charged higher rates when billed, resulting in complaints.

2.1.2 Testing Terminology

Testing is not debugging

In order to remedy a software fault it has to be located. To start with, we only know the effect of the fault, but not its location within the code. The process of finding and correcting faults is called debugging and is the responsibility of the developer. Debugging is often confused with testing, although these are two distinct and very different tasks. While debugging pinpoints software faults, testing is used to reveal the effect a fault causes (see figure 2-1).

Confirmation testing

Correcting a fault improves the quality of the product (assuming the correction doesn't cause additional, new faults). Tests used to check that a fault has been successfully remedied are called confirmation tests. Testers are often responsible for confirmation testing, whereas developers are more likely to be responsible for component testing (and debugging). However, these roles can change in an agile development environment or for other software lifecycle models.

Unfortunately, in real-world situations fault correction often leads to the creation of new faults that are only revealed when completely new input scenarios are used. Such unpredictable side effects make testing trickier. Once a fault has been corrected you need to repeat your previous tests to make sure the targeted failure has been remedied, and you also need to write new tests that check for unwanted side effects of the correction process.

Objectives of testing

Static and dynamic tests are designed to achieve various objectives:

- A qualitative evaluation of work products related to the requirements, the specifications, user stories, program design, and code
- Prove that all specific requirements have been completely implemented and that the test object functions as expected for the users and other stakeholders
- Provide information that enables stakeholders to make a solid estimate of the test object's quality and thus generate confidence in the quality provided[3]
- The level of quality-related risk can be reduced through identification and correction of software failures. The system will then contain fewer undiscovered faults.
- Analysis of the program and its documentation in order to avoid unwanted faults, and to document and remedy known ones
- Analyze and execute the program in order to reproduce known failures
- Receive information about the test object in order to decide whether the component in question can be committed for integration with other components
- Demonstrate that the test object adheres and/or conforms to the necessary contractual, legal and regulatory requirements and standards

Objectives depend on context

Test objectives can vary depending on the context. Furthermore, they can vary according to the development model you use (agile or otherwise) and the level of test you are performing—i.e., component, integration, system, or acceptance tests (see section 3.4).

When you are testing a component, your main objective should be to reveal as many failures as possible and to identify (i.e., debug) and remedy the underlying faults as soon as possible. Another primary objective can be to select tests that achieve the maximum possible level of code coverage (see section 2.3.1).

One objective of acceptance testing is to confirm that the system works and can be used as planned, and thus fulfills all of its functional and non-functional requirements. Another is to provide information that enables stakeholders to evaluate risk and make an informed decision about whether (or not) to go live.

3. If comprehensive testing reveals few (or no) failures, this increases stakeholder confidence in the product.

Side Note: Scheme for naming different types of testing

The various names used for different types of tests can be confusing. To understand the naming of tests it is useful to differentiate between the following naming categories:

1. **Test objective**
   The naming of a test type is based on the test objective (for example, a "load test").
2. **Test method/technique**
   A test is named according to the method or technique used to specify and/or perform the test (i.e., "state transition testing", as described in section 5.1.3)
3. **Test object**
   A test is named according to the type of object to be tested (for example, "GUI test" or "database test")
4. **Test level**
   A test is named according to the corresponding level of the development model being used (for example, a "system test")
5. **Test person**
   A test is named after the person or group who perform the test (for example, "developer test", "user test")
6. **Test scope**
   A test is named according to its scope (for example, a "partial regression test")

As you can see, not all of these terms define a distinct type of test. Instead, the different names highlight different aspects of a test that are important or in focus in a particular context or with regard to a particular testing objective.

2.1.3 Test Artifacts and the Relationships Between Them

The previous sections have already described some types of test artifacts. The following sections provide an overview of the types of artifacts necessary to perform dynamic testing.

Test basis

The test basis is the cornerstone of the testing process. As previously noted, the test basis comprises all documents that help us to decide whether a failure has occurred during testing. In other words, the test basis defines the expected behavior of the test object. Common sense and specialist knowledge can also be seen as part of the test basis and can be used to reach a decision. In most cases a requirements document, a specification, or a user story is available, which serves as a test basis.

Test cases and test runs

The test basis is used to define test cases, and a test run takes place when the test object is fed with appropriate test data and executed on a computer. The results of the test run are checked and the team decides

whether a failure has occurred—i.e., whether there is a discrepancy between the test object's expected and actual behaviors. Usually, certain preconditions have to be met in order to run a test case—for example, the corresponding database has to be available and filled with suitable data.

Test conditions

An individual test cannot be used to test the entire test basis, so it has to focus on a specific aspect. Test conditions are therefore extrapolated from the test basis in order to pursue specific test objectives (see above). A test condition can be checked using one or more tests and can be a function, a transaction, a quality attribute, or a structural element of a component or system. Examples of test conditions in our case study *VSR-II* system are vehicle configuration permutations (see section 5.1.5), the look and feel of the user interface, or the system's response time.

Test item

By the same token, a test object can rarely be tested as a complete object in its own right. Usually, we need to identify separate items that are then tested using individual test cases. For example, the test item for *VSR-II*'s price calculation test condition is the `calculate_price()` method (see section 5.1.1). The corresponding test cases are specified using appropriate testing techniques (see Chapter 5).

Test suites and test execution schedules

It makes little sense to perform test cases individually. Test cases are usually combined in test suites that are executed in a test cycle. The timing of test cycles is defined in the test execution schedule.

Test scripts

Test suites are automated using scripts that contain the test sequence and all of the actions required to create the necessary preconditions for testing, and to clean up once testing is completed. If you execute tests manually, the same information has to be made available for the manual tester.

Test logs

Test runs are logged and recorded in a test summary report.

Test plan

For every test object, you need to create a test plan that defines everything you need to conduct your tests (see section 6.2.1). This includes your choice of test objects and testing techniques, the definition of the test objectives and reporting scheme, and the coordination of all test-related activities.

Figure 2-2 shows the relationships between the various artifacts involved. Defining the individual activities involved in the testing process (see section 2.3) helps to clarify when each artifact is created.

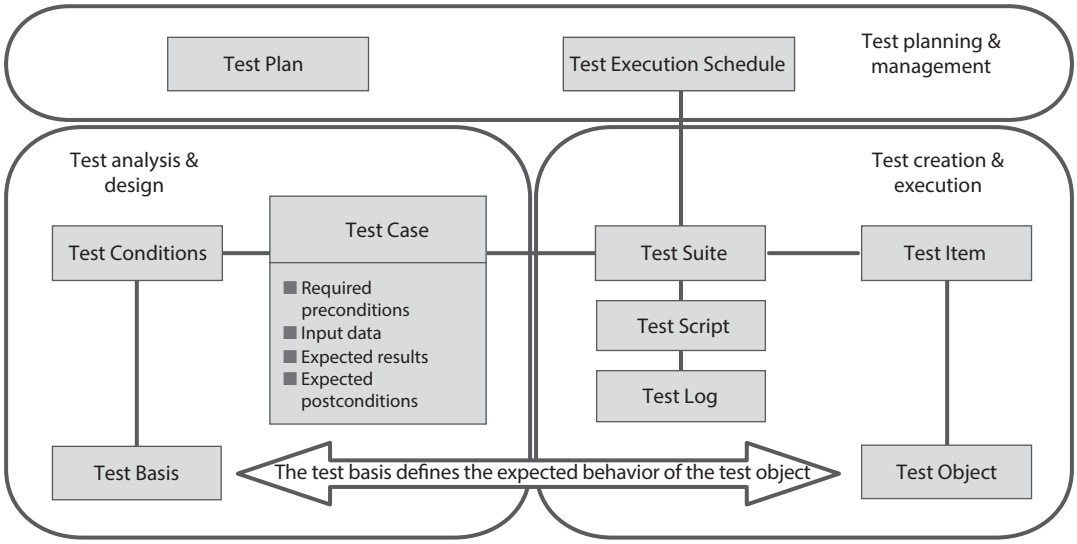

Fig. 2-2 *The relationships between test artifacts*

2.1.4 Testing Effort

Testing effort depends on the project (environment)

Testing takes up a large portion of the development effort, even if only a part of all conceivable tests—or, more precisely, all conceivable test cases—can be considered. It is difficult to say just how much effort you should spend testing, as this depends very much on the nature of the project at hand.[4]

The importance of testing—and thus the amount of effort required for testing—is often made clear by the ratio of testers to developers. In practice, the following ratios can be found: from one tester for every ten developers to three testers per developer. Testing effort and budget vary massively in real-world situations.

Case Study: Testing effort and vehicle variants

VSR-II enables potential customers to configure their own vehicle on a computer screen. The extras available for specific models and the possible combinations of options and preconfigured models are subject to a complex set of rules. The old *VSR* System allowed customers to select combinations that were not actually deliverable. As a consequence of the *VSR-II* QA/Test planning requirement *Functional suitability/DreamCar = high* (see below) customers should no longer be able to select non-deliverable combinations.

4. Section 6.2.5 goes into more detail on this topic.

The product owner responsible for the *DreamCar* module wants to know how much testing effort will be required to test this aspect of the module as comprehensively as possible. To do this, he makes an estimate of the maximum number of vehicle configuration options available. The results are as follows:

There are 10 vehicle models, each with 5 different engine options; 10 types of wheel rims with summer or winter tires; 10 colors, each with matt, glossy, or pearl effect options; and 5 different entertainment systems. These options result in $10\times5\times10\times2\times10\times3\times5=150.000$ different variants, so testing one variant every second would take a total of 1.7 days.

A further 50 extras (each of which is selectable or not) produce a total of 150.000×2^{50} = 168.884.986.026.393.600.000 variations.

The product owner intuitively knows that he doesn't have to test for every possible combination, but rather for the rules that define which combinations of options are not deliverable. Nevertheless, possible software faults create the risk that the *DreamCar* module wrongly classifies some configurations as deliverable (or permissible combinations as non-deliverable).

How much testing effort is required here and how much can it effectively cost? The product owner decides to ask the QA/testing lead for advice. One possible solution to the issue is to use pairwise testing (see the side note in section 5.1.5).

Side Note:
When is increased testing effort justifiable?

Is a high testing effort affordable and justifiable? Jerry Weinberg's response to this question is: "Compared with what?" [DeMarco 93]. This response points out the risks of using a faulty software system. Risk is calculated from the likelihood of a certain situation arising and the expected costs when it does. Potential faults that are not discovered during testing can later generate significant costs.

Example:
The cost of failure

In March 2016, a concatenation of software faults destroyed the space telescope *Hitomi*, which was built at a cost of several hundred million dollars. The satellite's software wrongly assumed that it was rotating too slowly and tried to compensate using countermeasures. The signals from the redundant control systems were then wrongly interpreted and the speed of rotation increased continuously until the centrifugal force became too much and Hitomi disintegrated (from [URL: Error Costs]).

In 2018 and 2019 two Boeing 737 MAX 8 airplanes crashed due to design flaws in the airplane's MCAS flight control software [URL: MAX-groundings]. Here too, the software—misdirected by incorrect sensor information—generated fatal countermeasures.

Testing effort has to remain in reasonable proportion to the results testing can achieve. "Testing makes economic sense as long as the cost of finding and remedying faults is lower than the costs produced by the corresponding failure occurring when the system is live."[5] [Pol 00]. Reasonable testing effort therefore always depends on the degree of risk involved in failure and an evaluation of the danger this incurs. The price of the destroyed space telescope Hitomi could have paid for an awful lot of testing.

Case Study: Risks and losses when failures occur

The *DreamCar* module constantly updates and displays the price of the current configuration. Registered customers with validated ID can order a vehicle online.

Once a customer clicks the *Order* button and enters their PIN, the vehicle is ordered and the purchase committed. Once the statutory revocation deadline has passed, the chosen configuration is automatically passed on to the production management system that initiates the build process.

Because the online purchase process is binding, if the system calculates and displays an incorrect price the customer has the right to insist on the paying that price. This means that wrongly calculated prices could lead to the manufacturer selling thousands of cars at prices that are too low. Depending on the degree of miscalculation, this could lead to millions of dollars in losses. Having each purchase order checked manually is not an option, as the whole point of the *VSR-II* system is that vehicles can be ordered completely automatically online.

Defining test thoroughness and scope depending on risk factors

Systems or system parts with a high risk have to be tested more extensively than those that do not cause major damage in case of failure.[6] Risk assessment has to be carried out for the individual system parts or even for individual failure modes. If there is a high risk of a system or subsystem malfunctioning, the test requires more effort than for less critical (sub) systems. These procedures are defined through international standards for the production of safety-critical systems. For example, the [RTC-DO 178B] Airborne Systems and Equipment Certification standard prescribes complex testing procedures for aviation systems.

Although there are no material risks involved, a computer game that saves scores incorrectly can be costly for its manufacturer, as such faults affect the public acceptance of a game and its parent company's other products, and can lead to lost sales and damage to the company's reputation.

5. These costs include not only software repair, renewed testing, and replacement of the faulty software, but also immaterial costs such as bad publicity and legal issues.
6. Section 6.2.4 goes into detail on this topic.