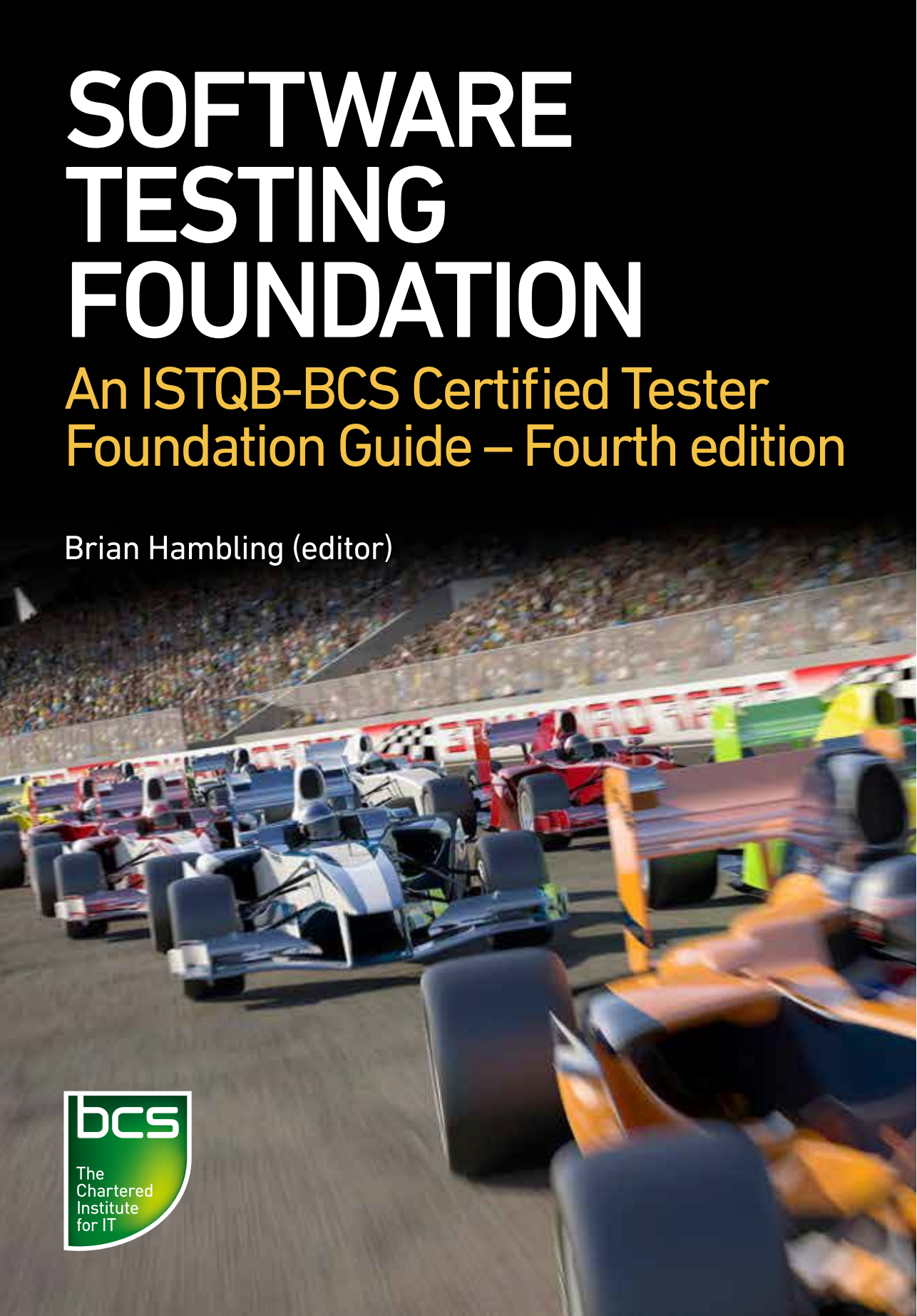
SOFTWARE TESTING FOUNDATION

An ISTQB-BCS Certified Tester Foundation Guide – Fourth edition

Brian Hambling (editor)



bcs
The Chartered Institute for IT

SOFTWARE TESTING

BCS, THE CHARTERED INSTITUTE FOR IT

BCS, The Chartered Institute for IT, is committed to making IT good for society. We use the power of our network to bring about positive, tangible change. We champion the global IT profession and the interests of individuals, engaged in that profession, for the benefit of all.

Exchanging IT expertise and knowledge
The Institute fosters links between experts from industry, academia and business to promote new thinking, education and knowledge sharing.

Supporting practitioners
Through continuing professional development and a series of respected IT qualifications, the Institute seeks to promote professional practice tuned to the demands of business. It provides practical support and information services to its members and volunteer communities around the world.

Setting standards and frameworks
The Institute collaborates with government, industry and relevant bodies to establish good working practices, codes of conduct, skills frameworks and common standards. It also offers a range of consultancy services to employers to help them adopt best practice.

Become a member
Over 70,000 people including students, teachers, professionals and practitioners enjoy the benefits of BCS membership. These include access to an international community, invitations to a roster of local and national events, career development tools and a quarterly thought-leadership magazine. Visit www.bcs.org/membership to find out more.

Further Information
BCS, The Chartered Institute for IT,
First Floor, Block D,
North Star House, North Star Avenue,
Swindon, SN2 1FA, United Kingdom.
T +44 (0) 1793 417 424
F +44 (0) 1793 417 444
(Monday to Friday, 09:00 to 17:00 UK time)
www.bcs.org/contact
http://shop.bcs.org/

SOFTWARE TESTING

An ISTQB–BCS Certified Tester Foundation guide
Fourth edition

Brian Hambling (editor), Peter Morgan, Angelina Samaroo, Geoff Thompson and Peter Williams

bcs
The Chartered Institute for IT

Published by BCS Learning and Development Ltd, a wholly owned subsidiary of BCS, The Chartered Institute for IT, First Floor, Block D, North Star House, North Star Avenue, Swindon, SN2 1FA, UK.
www.bcs.org

Paperback ISBN: 978-1-78017-492-1
PDF ISBN: 978-1-78017-493-8
ePUB ISBN: 978-1-78017-494-5
Kindle ISBN: 978-1-78017-495-2

British Cataloguing in Publication Data.
A CIP catalogue record for this book is available at the British Library.

Publisher's acknowledgements
Publisher: Ian Borthwick
Commissioning editor: Rebecca Youé
Production manager: Florence Leroy
Project manager: Sunrise Setting Ltd
Copy-editor: Denise Bannerman
Proofreader: Barbara Eastman
Indexer: Matthew Gale
Cover design: Alex Wright
Picture Credits: mevans
Typeset by Lapiz Digital Services, Chennai, India.

CONTENTS

LIST OF FIGURES AND TABLES

AUTHORS

Brian Hambling has experienced software development from a developer's, project manager's and quality manager's perspective in a career spanning over 35 years. He has worked in areas as diverse as real-time avionics, legacy systems maintenance and ebusiness strategies. He contributed to the development of software quality standards while at the Ministry of Defence and later became the head of systems and software engineering at The University of Greenwich. He was technical director of ImagoQA and general manager of Microgen IQA, a specialist company providing consultancy in software testing and quality assurance primarily to the financial services sector. He is now concentrating on writing.

Peter Morgan was a freelance testing practitioner, now mainly retired. He worked as a hands-on tester for a number of years, often on large projects. He has worked in many business sectors in the UK and has been involved with the examination side of testing since taking the Foundation Certificate in 2001. Peter still writes exam questions and regularly reviews course submissions of testing courses on behalf of the UKTB, looking for syllabus compliance and to see that potential students are well prepared for their own exams.

Angelina Samaroo has a degree in Aeronautical Engineering from Queen Mary University of London. She is a Chartered Engineer and Fellow of the Institution of Engineering and Technology (IET). She spent 10 years working on the mission software for the Tornado ADV fighter jet. During this time, she took an interest in the career development of new engineers and led the company graduate development scheme. She then joined a small consultancy in software testing, specialising in providing training to test professionals worldwide. She now works for Pinta Education Limited and provides training in software testing, business analysis and programming.

Geoff Thompson is the UK Director of Testing Services for Planit Testing, part of the global Planit Testing group. In this role he is able to champion his passion for software testing, test management and process improvement. He is a founder member of the International Software Testing Qualification Board (ISTQB), the TMMi Foundation, and the UK Testing Board and is currently the Vice President of the ISTQB and Chairman of the UK Testing Board. He co-authored the BCS book Software Testing - An ISEB/ISTQB foundation and is a recognized international speaker, keynoting in many conferences, and was the chair of EuroSTAR 2011. He is a founding member and chairman of the TMMi Foundation (see www.tmmifoundation.org.uk) In 2008 Geoff was awarded the European Testing Excellence Award, and in 2015 he was awarded the Software Testing European Lifetime Achievement award.

Peter Williams previously worked in methods and systems improvement before moving into systems development and subsequently software testing. He has been a self-employed contract test manager or consultant in both financial services and the public sector. He has evaluated test processes and subsequently implemented improvements, at various organisations, including test management and execution tools as appropriate. He has an MSc in computing from the Open University and was chairman of the Examinations Panel for the ISEB Foundation Certificate in Software Testing.

ABBREVIATIONS

ALM	Application Life Cycle Management
API	Application Program Interface
ATDD	Acceptance Test Driven Development
AUT	Application Under Test
BACS	Bankers Automated Clearing Services
BDD	Behaviour Driven Development
CFG	Control Flow Graph
COTS	Commercial Off-the-Shelf
CTFL	Certified Tester Foundation Level
DOS	Denial of Service
DSL	Domain-Specific Language
GUI	Graphical User Interface
ISEB	Information Systems Examination Board
ISTQB	International Software Testing Qualifications Board
MISRA	Motor Industry Software Reliability Association
RUP	Rational Unified Process
SDLC	Software Development Life Cycle
SIGiST	Specialist Interest Group in Software Testing
SQL	Structured Query Language
ST	State Table
SUT	System Under Test
TDD	Test Driven Development
UML	Unified Modeling Language
XML	Extensible Markup Language

PREFACE

When I started work on the first edition of this book in 2006, I little thought that I would be drafting a preface to a third edition in 2015. My fellow authors and I have been amazed at the continuing success of this simple guide to understanding the ISTQB Foundation Certificate in Software Testing and passing the exam, yet feedback from readers who have passed the exam by using this book as their sole preparation has been steadily amassing.

In this fourth edition, we have tried to provide some new material and even better exam preparation resources. Many of the exercises, examples and sample questions have been updated to reflect the way the exam has evolved over the years, and for this edition we have put together a complete mock examination. This gives commentary to explain the correct answer and why each of the distracters is incorrect. Moreover, as a mock exam, the candidates can time themselves to check they are prepared for the real thing. We think that this is a major improvement on previous editions, and we hope you will find it really useful.

INTRODUCTION

The Foundation Certificate in Software Testing was introduced as part of the BCS Professional Certification portfolio (formerly ISEB) in 1998; since then, over 550,000 Foundation Certificates have been awarded. An Intermediate Level Certificate was introduced in 2007 as a step towards the more advanced Practitioner Certificates. (Visit www.bcs.org/certifications for more information.)

The International Software Testing Qualifications Board (ISTQB) (www.istqb.org) was set up in 2001 to offer a similar certification scheme to as many countries as wished to join this international testing community. The UK was a founding member of ISTQB and, in 2005, adopted the ISTQB Foundation Certificate syllabus as the basis of examinations for the Foundation Certificate in the UK. The Foundation Certificate is now an entry qualification for the ISTQB Advanced Certificate. The Certified Tester Foundation Level (CTFL) syllabus has been updated and released in a 2018 version, and this book relates to the 2018 version of the syllabus.

This book has been written specifically to help potential candidates for the ISTQB–BCS CTFL examination. The book is therefore structured to support learning of the key ideas in the syllabus quickly and efficiently for those who do not plan to attend a course, and to support structured revision for anyone preparing for the exam.

In this introductory chapter, we will explain the nature and purpose of the Foundation Level and provide an insight into the way the syllabus is structured and the way the book is structured to support learning in the various syllabus areas. Finally, we offer guidance on the best way to use this book, either as a learning resource or as a revision resource.

PURPOSE OF FOUNDATION

The CTFL Certificate is the first level of a hierarchy of ISTQB–BCS certificates in software testing, and leads naturally into the next level, known as the BCS Intermediate Certificate in Software Testing, which in turn leads on to the ISTQB Advanced Level, followed by the various ISTQB Expert Level examinations.

The Foundation Level provides a very broad introduction to the whole discipline of software testing. As a result, coverage of topics is variable, with some only briefly mentioned and others studied in some detail. The arrangement of the syllabus and the required levels of understanding are explained in the next section.

The authors of the syllabus have aimed it at people with various levels of experience in testing, including those with no experience at all. This makes the certificate accessible to those who are or who aim to be specialist testers, but also to those who require a more general understanding of testing, such as project managers and software development managers. One specific aim of this qualification is to prepare certificate holders for the next level of certification, but the Foundation Level has sufficient breadth and depth of coverage to stand alone.

THE CERTIFIED TESTER FOUNDATION LEVEL SYLLABUS

Syllabus content and structure

The syllabus is broken down into six main sections, each of which has associated with it a minimum contact time that must be included within any accredited training course:

1. Fundamentals of testing (175 minutes).
2. Testing throughout the Software Development Life Cycle (100 minutes).
3. Static testing (135 minutes).
4. Test techniques (330 minutes).
5. Test management (225 minutes).
6. Tool support for testing (40 minutes).

The relative timings are a reliable guide to the amount of time that should be spent studying each section of the syllabus.

Each section of the syllabus also includes a list of learning objectives that provides candidates with a guide to what they should know when they have completed their study of a section and a guide to what can be expected to be asked in an examination. The learning objectives can be used to check that learning or revision is adequate for each topic. In the book, which is structured around the syllabus sections, we have presented the learning objectives for each section at the beginning of the relevant chapter, and the summary at the end of each chapter confirms how those learning objectives have been addressed.

Finally, each topic in the syllabus has associated with it a level of understanding, represented by the legend K1, K2 or K3:

- Level of understanding K1 is associated with recall, so that a topic labelled K1 contains information that a candidate should be able to remember but not necessarily use or explain.
- Level of understanding K2 is associated with the ability to explain a topic or to classify information or make comparisons.
- Level of understanding K3 is associated with the ability to apply a topic in a practical setting.

The level of understanding influences the level and type of questions that can be expected to be asked about that topic in the examination. More detail about the question style and about the examination is given in **Chapter 7**. Example questions, written to the level and in the formats used in the examination, are included within each chapter to provide generous examination practice, and there is a complete practice examination paper in **Appendix A**, with answers and an explanation of the correct response to each question.

Syllabus updates and changes

The syllabus was last updated in 2018 and the book is in line with the changes and additions introduced in that version of the syllabus. An extension to the Foundation syllabus has also been introduced to cover a significant aspect of software testing that has grown in prominence in recent years. The Foundation Level Extension – Agile Tester is a complete new syllabus with its own separate examination and that syllabus is not addressed in this book.

Throughout this book you will find references to different testing standards that form reference points for parts of the ISTQB Software Testing Foundation syllabus.

In the 2018 syllabus and in this book:

- ISO/IEC/IEEE 29119 replaces IEEE Standard 829.
- ISO/IEC 25010 replaces ISO 9126.
- ISO/IEC 20246 replaces IEEE 1028.

ISO/IEC/IEEE 29119 *Software testing* is an internationally agreed set of standards for software testing that can be used within any Software Development Life Cycle or organisation. There are currently five standards within ISO/IEC/IEEE 29119:

- ISO/IEC 29119-1: *Concepts & definitions* (published September 2013) facilitates understanding and use of all other standards and the vocabulary used within the 29119 series.
- ISO/IEC 29119-2: *Test processes* (published September 2013) defines a generic process model for software testing that can be used within any Software Development Life Cycle.
- ISO/IEC 29119-3: *Test documentation* (published September 2013) defines templates for test documentation that cover the entire software testing life cycle.
- ISO/IEC 29119-4: *Test techniques* (published December 2015) defines software test design techniques (also known as test case design techniques or test methods).
- ISO/IEC 29119-5: *Keyword-driven testing* (published November 2016) defines an international standard for supporting keyword-driven testing.

RELATIONSHIP OF THE BOOK TO THE SYLLABUS

The book is structured into chapters that mirror the sections of the syllabus so that you can work your way through the whole syllabus or select topics that are of particular interest or concern. The structure enables you to go straight to the place you need, with confidence either that what you need to know will be covered there and nowhere else, or that relevant cross references will be provided.

Each chapter of the book incorporates the learning objectives from the syllabus and identifies the required level of understanding for each topic. Each chapter also includes examples of typical examination questions to enable you to assess your current knowledge of a topic before you read the chapter, and further questions at the end of each chapter to provide practice in answering typical examination questions. Topics requiring a K3 level of understanding are presented with worked examples as a model for the level of application expected from real examination questions. Answers are provided for all questions, and the rationale for the correct answer is discussed for all practice questions.

A final chapter explains the Foundation Level examination strategy and provides guidance on how to prepare for the examination and how to manage the examination experience to maximise your own performance.

HOW TO GET THE BEST OUT OF THIS BOOK

This book is designed for use by different groups of people. If you are using the book as an alternative to attending an accredited course, you will probably find the first method of using the book described below to be of greatest value. If you are using the book as a revision aid, you may find the second approach more appropriate. In either case, you would be well advised to acquire a copy of the syllabus and a copy of the sample examination paper (both available free from www.istqb.org) as reference documents, though neither is essential and the book stands alone as a learning and revision aid.

Using the book as a learning aid

For those of you using the book as an alternative to attending an accredited course the first step is to familiarise yourself with the syllabus structure and content by skim reading the opening sections of each chapter, where the learning objectives are identified for each topic. You may then find it helpful to turn to **Chapter 7** and become familiar with the structure of the examination and the types and levels of questions that you can expect in the examination. From here you can then work through each of the six main chapters in any sequence before returning to **Chapter 7** to remind yourself of the main elements of the examination.

For each chapter begin by attempting the self-assessment questions at the beginning to get initial confirmation of your level of confidence in the topics covered by that chapter. This may help you to prioritise how you spend your time. Work first through the chapters where your knowledge is weakest, attempting all the exercises and following through all the worked examples. Read carefully through the chapters where your knowledge is

less weak, but still not good enough to pass the exam. You can be more selective with exercises and examples here, but make sure you attempt the practice questions at the ends of the chapters. For the areas where you feel strong you can use the chapter for revision but remember to attempt the practice questions to confirm positively your initial assessment of your level of knowledge. Each chapter contains 'checks of understanding' between sections so that you can determine whether you have picked up the key points from that section. These are important; if you find you cannot answer them, you would be wise to go back over the material to revise anything that you did not really absorb at the first read. There is also a summary section at the end of each chapter that reiterates the learning objectives, so reading the first and last sections of a chapter will help you to understand how your current level of knowledge relates to the level required to pass the examination. The best confirmation of this is to attempt questions at the appropriate K level for each topic; these are provided throughout the book and there is a complete mock examination at the end so that you can really test whether you are ready for the exam.

Using the book as a revision aid

If you are using this book for final revision, perhaps after completing an accredited course, you might like to begin by using a selection of the example questions at the end of each chapter as a 'revision mock examination'. **Appendix A1** contains a complete mock exam, with all the answers in **Appendix A2**, which will provide some experience of answering typical questions under the same time pressures that you will experience in the real examination, and this will provide you with a fairly reliable guide to your current state of readiness to take the real examination. You can also discover which areas most need revision from your performance in the mock exam, and this will guide you as you plan your revision. There is a complete question-by-question commentary on the mock exam in **Appendix A3** so that you can identify why you got any questions wrong.

Revise first where you feel weakest. You can use the opening sections of each chapter, containing the learning objectives and the self-assessment questions, together with the 'Check of understanding' provided at the end of each section, and the chapter summary at the end of each chapter, to refine further your awareness of your own weaknesses. From here you can target your studies very accurately. Remember that every K3 topic will have at least one worked example and some exercises to help you build your confidence before tackling questions at the level set in the real examination.

You can get final confirmation of your readiness to sit the real examination by taking the sample examination paper available from BCS.

1 THE FUNDAMENTALS OF TESTING

Peter Morgan

If you were buying a new car, you would not expect to take delivery from the showroom with a scratch down the side of the vehicle. The car should have the right number of wheels (including a 'spare' for emergencies), a steering wheel, an engine and all the other essential components, and it should come with appropriate documentation, with all pre-sales checks completed and passed satisfactorily. The car you receive should be the car described in the sales literature; it should have the correct engine size, the correct colour scheme and whatever extras you have ordered, and performance in areas such as fuel consumption and maximum speed should match published figures. In short, a level of expectation is set by brochures, by your experience of sitting in the driving seat and probably by a test drive. If your expectations are not met, you will feel justifiably aggrieved.

This kind of expectation seems not to apply to new software installations; examples of software being delivered not working as expected, or not working at all, are common. Why is this? There is no single cause that can be rectified to solve the problem, but one important contributing factor is the inadequacy of the testing to which software applications are exposed.

Software testing is neither complex nor difficult to implement, yet it is a discipline that is seldom applied with anything approaching the necessary rigour to provide confidence in delivered software. Software testing is costly in human effort or in the technology that can multiply the effect of human effort, yet it is seldom implemented at a level that will provide any assurance that software will operate effectively, efficiently or even correctly.

This book explores the fundamentals of this important but neglected discipline to provide a basis on which a practical and cost-effective software testing regime can be constructed.

INTRODUCTION

In this opening chapter we have three very important objectives to achieve. First, we will introduce you to the fundamental ideas that underpin the discipline of software testing, and this will involve the use and explanation of some new terminology. Second, we will establish the structure that we have used throughout the book to help you to use the book as a learning and revision aid. Third, we will use this chapter to point forward to the content of later chapters.

The key ideas of software testing are applicable irrespective of the software involved and any particular development methodology (waterfall, Agile etc.). Software development methodologies are discussed in detail in **Chapter 2.**

We begin by defining what we expect you to get from reading this chapter. The learning objectives below are based on those defined in the Software Foundation Certificate syllabus, so you need to ensure that you have achieved all of these objectives before attempting the examination.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions immediately after the learning objectives listed below, the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

What is testing? (K2)

- FL-1.1.1 Identify typical objectives of testing. (K1)
- FL-1.1.2 Differentiate testing from debugging.

Why is testing necessary? (K2)

- FL-1.2.1 Give examples of why testing is necessary.
- FL-1.2.2 Describe the relationship between testing and quality assurance and give examples of how testing contributes to higher quality.
- FL-1.2.3 Distinguish between error, defect, and failure.
- FL-1.2.4 Distinguish between the root cause of a defect and its effects.

Seven testing principles (K2)

- FL-1.3.1 Explain the seven testing principles.

Test process (K2)

- FL-1.4.1 Explain the impact of context on the test process.
- FL-1.4.2 Describe the test activities and respective tasks within the test process.
- FL-1.4.3 Differentiate the work products that support the test process.
- FL-1.4.4 Explain the value of maintaining traceability between the test basis and test work products.

The psychology of testing (K2)

- FL-1.5.1 Identify the psychological factors that influence the success of testing. (K1)
- FL-1.5.2 Explain the difference between the mindset required for test activities and the mindset required for development activities.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K2)
Which of the following correctly describes the interdependence between an error, a defect and a failure?

a. An error causes a failure that can lead to a defect.
b. A defect causes a failure that can lead to an error.
c. An error causes a defect that can lead to a failure.
d. A failure causes an error that can lead to a defect.

Question SA2 (K2)
An online site where goods can be bought and sold has been implemented. Which one of the following could be the root cause of a defect?

a. Customers complain that the time taken to move to the 'payments' screen is too long.
b. The lead business analyst was not familiar with all possible permutations of customer actions.
c. Multiple customers can buy the **one** item that is for sale.
d. There is no project-wide defect tracking system in use.

Question SA3 (K2)
Which of the following illustrates the principle of defect clustering?

a. Testing everything in most cases is not possible
b. Even if no defects are found, testing cannot show correctness.
c. It is incorrect to assume that finding and fixing a large number of defects will ensure the success of a system.
d. A small subset of the code will usually contain most of the defects discovered during the testing phases.

WHY SOFTWARE FAILS

Examples of software failure are depressingly common. Here are some you may recognise:

- After successful test flights and air worthiness accreditation, problems arose in the manufacture of the Airbus A380 aircraft. Assembly of the large subparts into the completed aircraft revealed enormous cabling and wiring problems. The wiring of large subparts could not be joined together. It has been estimated that the direct or indirect costs of rectification were $6.1 billion. (Note: this problem was quickly fixed and the aircraft entered into service within 18 months of the cabling difficulties being identified.)
- When the UK Government introduced online filing of tax returns, a user could sometimes see the amount that a previous user earned. This was regardless of the physical location of the two applicants.
- In November 2005, information on the UK's top 10 wanted criminals was displayed on a website. The publication of this information was described in newspapers and on morning radio and television and, as a result, many people attempted to access the site. The performance of the website proved inadequate under this load and it had to be taken offline. The publicity created performance peaks beyond the capacity of the website.
- A new smartphone mapping application (app) was introduced in September 2012. Among many other problems, a museum was incorrectly located in the middle of a river, and Sweden's second city, Gothenburg, seemed to have disappeared from at least one map.
- Security breaches at the US military resulted in the payment details of many personnel (perhaps even 'all') being compromised, including names, addresses, email addresses and bank details.

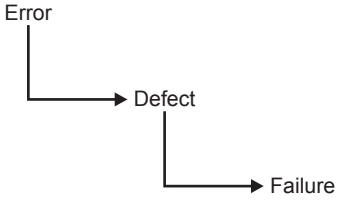
Perhaps these examples are not the same as the notorious final payment demands for 'zero pounds and zero pence' of some utilities customers in the 1970s. However, what is it that still makes them so startling? Is it a sense that something fairly obvious was missed? Is it the feeling that, expensive and important as they were, the systems were allowed to enter service before they were ready? Do you think these systems were adequately tested? Obviously, they were not but in this book we want to explore why this was the case and why these kinds of failure continue to plague us.

To understand what is going on we need to start at the beginning, with the people who design systems. Do they make mistakes? Of course they do. People make mistakes because they are fallible, but there are also many pressures that make mistakes more likely. Pressures such as deadlines, complexity of systems and organisations, and changing technology all bear down on designers of systems and increase the likelihood of defects in specifications, in designs and in software code. Errors are where major system failures usually begin. An error is best thought of as 'invisible', or better perhaps as intangible – you cannot touch it. It is an incorrect thought, a wrong assumption, or a thing that is forgotten, or not considered. Only when something is written down can it become 'a fault' or a defect. So, an incorrect choice can lead to a document with a defect in it, which, if it is used to specify a component, can result in the component being faulty

and this may exhibit incorrect behaviour. If this faulty component is built into a system, the system may fail. While failure is not always guaranteed, it is likely that errors in the thought processes as specifications are produced will lead to faulty components and faulty components will cause system failure.

An error (or mistake) leads to a defect (or fault), which can cause an observed failure (Figure 1.1).

Figure 1.1 Effect of an error



There are other reasons why systems fail. Environmental conditions such as the presence of radiation, magnetism, electronic fields or pollution can affect the operation of hardware and firmware and lead to system failure.

It is worth pointing out at this stage that not every apparent failure is a real failure – something appears to be a failure in the software, but the observed behaviour is correct. Perhaps the tester that created the test misunderstood what should happen in the precise circumstances. When an apparent failure in test is actually the application or system performing correctly, this is termed a false positive. Conversely, a false negative is where there is a real failure, but this is not identified as such and the test is seen as correct.

If we want to avoid failure, we must either avoid errors and faults or find them and rectify them. Testing can contribute to both avoidance and rectification, as we will see when we have looked at the testing process in a little more detail. One thing is clear: if we wish to identify errors through testing we need to begin testing as soon as we begin making errors – right at the beginning of the development process – and we need to continue testing until we are confident that there will be no serious system failures – right at the end of the development process.

Before we move on, let us just remind ourselves of the importance of what we are considering. Incorrect software can harm:

- people (e.g. by causing an aircraft crash in which people die, or by causing a hospital life support system to fail);
- companies (e.g. by causing incorrect billing, which results in the company losing money);
- the environment (e.g. by releasing chemicals or radiation into the atmosphere).

Software failures can sometimes cause all three of these at once. The scenario of a train carrying nuclear waste being involved in a crash has been explored to help build public confidence in the safety of transporting nuclear waste by train. A failure of the train's on-board systems, or of the signalling system that controls the train's movements, could lead to catastrophic results. This may not be likely (we hope it is not) but it is a possibility that could be linked with software failure. Software failures, then, can lead to:

- loss of money;
- loss of time;
- loss of business reputation;
- injury;
- death.

KEEPING SOFTWARE UNDER CONTROL

With all of the examples we have seen so far, what common themes can we identify? There may be several themes that we could draw out of the examples, but one theme is clear: either insufficient testing or the wrong type of testing was done. More and better software testing seems a reasonable aim, but that aim is not quite as simple to achieve as we might expect.

Exhaustive testing of complex systems is not possible

The launch of the smartphone app occurred at the same time as a new phone hardware platform – the new app was only available on the new hardware for what many would recognise as the market leader at that time. The product launch received extensive worldwide coverage, with the mapping app given a prominent place in launch publicity. In a matter of hours there were tens of thousands of users, and numbers quickly escalated, with many people wanting to see their location in the new app and see how this compared with (for example) Google Maps. Each phone user was an expert in his/her location – after all they lived there, and 'test cases' were generated showing that problems existed.

If every possible test had been run, problems would have been detected and rectified prior to the product launch. However, if every test had been run, the testing would still be running now, and the product launch would never have taken place; this illustrates one of the general principles of software testing, which are explained below. With large and complex systems, it will never be possible to test everything exhaustively; in fact, it is impossible to test even moderately complex systems exhaustively.

For the mapping app, it would be unhelpful to say that not enough testing was done; for this particular project, and for many others of similar complexity, that would certainly always be the case. Here the problem was that the right sort of testing was not done because the problems had not been detected.

Testing and risk

Risk is inherent in all software development. The system may not work or the project to build it may not be completed on time, for example. These uncertainties become more significant as the system complexity and the implications of failure increase. Intuitively, we would expect to test an automatic flight control system more than we would test a video game system. Why? Because the risk is greater. There is a greater probability of failure in the more complex system and the impact of failure is also greater. What we test, and how much we test it, must be related in some way to the risk. Greater risk implies more and better testing.

There is much more on risk and risk management in **Chapter 5**.

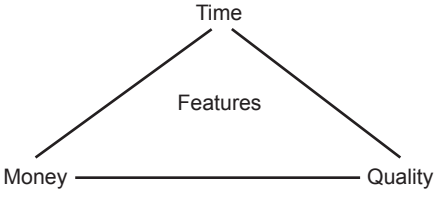
Testing and quality

Quality is notoriously hard to define. If a system meets its users' requirements, that constitutes a good starting point. In the examples we looked at earlier, the online tax returns system had an obvious functional weakness in allowing one user to view another user's details. While the user community for such a system is potentially large and disparate, it is hard to imagine any user that would find that situation anything other than unacceptable. In the top 10 criminals example, the problem was slightly different. There was no failure of functionality in this case; the system was simply swamped by requests for access. This is an example of a non-functional failure, in that the system was not able to deliver its services to its users because it was not designed to handle the peak load that materialised after radio and TV coverage.

The problems with the Airbus A380 aircraft is an interesting story, because although completed subparts could not be brought together to build an entire aircraft, each of the subparts was 'correctly manufactured'. Aircraft are increasingly sophisticated, and the A380 aircraft has approximately 530 km of cables, 100,000 wires and 40,300 connectors. Software is used both to design the aircraft and in the manufacture. However, the large subparts were made in two different counties, with different versions of the software in each manufacturing base. So, when Airbus was bringing together two halves of the aircraft, the different software meant that the wiring on one did not match the wiring on the other. The cables could not meet up without being changed. Testing may have taken place, but it did not test something as straightforward as the versions of the design software and whether they were compatible (which in this case they were plainly not). Each large subpart was built according to its own version of the CATIA (Computer Aided Three-Dimensional Interactive Application) software. The result did not give an aircraft that could fly.

Of course, the software development process, like any other, must balance competing demands for resources. If we need to deliver a system faster (i.e. in less time), for example, it will usually cost more. The items at the corners (or vertices) of the triangle of resources in Figure 1.2 are time, money and quality. These three items affect one another, and also influence the features that are or are not included in the delivered software.

Figure 1.2 Resources triangle



One role for testing is to ensure that key functional and non-functional requirements are examined before the system enters service and any defects are reported to the development team for rectification. Testing cannot directly remove defects, nor can it directly enhance quality. By reporting defects, it makes their removal possible and so contributes to the enhanced quality of the system. In addition, the systematic coverage of a software product in testing allows at least some aspects of the quality of the software to be measured. Testing is one component in the overall quality assurance activity that seeks to ensure that systems enter service without defects that can lead to serious failures.

Deciding when 'enough is enough'

How much testing is enough, and how do we decide when to stop testing?

We have so far decided that we cannot test everything, even if we would wish to. We also know that every system is subject to risk of one kind or another and that there is a level of quality that is acceptable for a given system. These are the factors we will use to decide how much testing to do.

The most important aspect of achieving an acceptable result from a finite and limited amount of testing is prioritisation. Do the most important tests first so that at any time you can be certain that the tests that have been done are more important than the ones still to be done. Even if the testing activity is cut in half, it will still be true that the most important testing has been done. The most important tests will be those that test the most important aspects of the system: they will test the most important functions as defined by the users or sponsors of the system, and the most important non-functional behaviour, and they will address the most significant risks.

The next most important aspect is setting criteria that will give you an objective test of whether it is safe to stop testing, so that time and all the other pressures do not confuse the outcome. These criteria, usually known as completion criteria, set the standards for the testing activity by defining areas such as how much of the software is to be tested (this is covered in more detail in **Chapter 4**) and what levels of defects can be tolerated in a delivered product (which is covered in more detail in **Chapter 5**).

Priorities and completion criteria provide a basis for planning (which will be covered in **Chapter 2** and **Chapter 5**) but the triangle of resources in Figure 1.2 still applies. In the end, the desired level of quality and risk may have to be compromised, but our approach ensures that we can still determine how much testing is required to achieve the agreed

levels and we can still be certain that any reduction in the time or effort available for testing will not affect the balance – the most important tests will still be those that have already been done whenever we stop.

CHECK OF UNDERSTANDING

1. Describe the interaction between errors, defects and failures.
2. Software failures can cause losses. Give three consequences of software failures.
3. What are the vertices of the 'triangle of resources'?

WHAT TESTING IS AND WHAT TESTING DOES

So far, we have worked with an intuitive idea of what testing is. We have recognised that it is an activity used to reduce risk and improve quality by finding defects, which is all true. There are indeed many different definitions of 'testing' when applied to software. Here is one that we have found useful – but don't worry: you are not expected to remember it!

> Testing is the systematic and methodical examination of a work product using a variety of techniques, with the express intention of attempting to show that it does not fulfil its desired or intended purpose. This is undertaken in an environment that represents most nearly that which will be used in live operation.

Like other definitions of testing, it is not perfect. But it does point the way to material that will be covered in later chapters of this book, including:

- systematic – it has to be planned (**Chapter 5**);
- methodical – it is a process (throughout, but initially later in this chapter);
- work product – it is not just code that is examined (**Chapter 3**);
- variety of techniques (**Chapter 4**).

Any definition of testing has limitations! The definition depends upon the aim(s) or goals of that testing, or indeed the testing of that application. For not all testing has the same aim, and the aim of testing can vary through the Software Development Life Cycle.

So, what does testing aim to do? Here are some of the principle reasons:

- to examine work products (note it is 'work product'; as well as code, this can include requirements, user stories and the overall design, amongst other items);
- to check if all the requirements have been satisfied;
- to see whether the item under test is complete, and works as the users and other stakeholders expect;

- to instil confidence in the quality of the item under test;
- to prevent defects – testing can not only 'catch' defects, but sometimes stop them happening in the first place;
- to find failures and defects and prevent these reaching the production version of the software. Often this is the **first** item that people will list;
- to give sufficient information to enable decision makers to make decisions, perhaps about whether the software product is suitable for release;
- to reduce the level of risk of inadequate software quality (e.g. previously undetected failures occurring in operation);
- to comply with contractual, legal or regulatory requirements or standards. Some standards require, for example, 80 per cent decision coverage to be shown by testing.

Testing can therefore be a multifaceted activity. However, we need to understand a little more about how software testing works in practice before we can think about how to implement effective testing.

Before that though, we will sum up what testing is, by looking at a few of the key terms or definitions that are relevant to this opening chapter. Remember, the keywords that are given in the syllabus for this section of material can be used in examination questions, where a clear understanding of the meaning of the terms is required. Testing relies upon an understanding of what the system, application or utility is meant to achieve. This is usually, but not always, written down, perhaps in a requirements document, in a description of the house style of web applications in the company, or in the interface definitions that are required for external systems. This body of knowledge is termed the **test basis**; testing has the test basis as a key starting point for what to test and how to test. Within the test basis, there are descriptions of what should happen under certain circumstances – 'if the user-name and password combination is incorrect, display an error message, and request that the operator tries again, and if the details are incorrect on the third attempt, lock the user account'. Such descriptions are called **test conditions** – what will happen as a consequence of previous choices or actions. Test conditions give the circumstances, but not usually the values required to run a test. The user-name or password may indeed be incorrect, but the specific values to be used are given in a **test case**. A test case is derived from one or more test conditions and describes in some detail what is required to enter into the application or system, and most importantly, what the expected outcome is, so that we know whether the test is successful or not (has 'passed' or 'failed'). In many instances, there are preconditions that must be in place before one or more particular test cases can be run and actions that are to be done when the test case or cases have been run. This is called a **test procedure**. Here are the definitions of these four terms, taken from the ISTQB glossary

Test basis:	The body of knowledge used as the basis for test analysis and design.
Test condition:	An aspect of the test basis that is relevant in order to achieve specific test objectives.

Test case:	A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.
Test procedure:	A sequence of test cases in execution order, and any associated actions that may be required to set up the initial pre-conditions and any wrap up activities post execution.

Testing and debugging

Testing and debugging are different kinds of activity, both of which are very important. Debugging is the process that developers go through to identify the cause of bugs, or defects in code, and undertake corrections. Ideally, some check of the correction is made, but this may not extend to checking that other areas of the system have not been inadvertently affected by the correction. Testing, on the other hand, is a systematic exploration of a component or system with the main aim of finding and reporting defects. Testing does not include correction of defects – these are passed on to the developer to correct. Testing does, however, ensure that changes and corrections are checked for their effect on other parts of the component or system.

Usually, developers will have undertaken some initial testing (and taken appropriate corrective action as necessary – debugging) to raise the level of quality of the component or system to a level that is worth testing; that is, a level that is sufficiently robust to enable rigorous testing to be performed. Debugging does not give confidence that the component or system meets its requirements completely. Testing makes a rigorous examination of the behaviour of a component or system and reports all defects found for the development team to correct. Testing then repeats enough tests to ensure that defect corrections have been effective. So, both are needed to achieve a quality result.

While it is generally true that testers test, and developers undertake debugging action, this is not always so clear-cut. In some software development methodologies (typically Agile development, but not restricted to this), testers are routinely involved in both component testing and debugging activities.

Defects, effects and root causes

We talked earlier about the differences and interconnections between an error, a defect and a failure. There is a similar relationship between the root cause of a defect, a description of the defect and the effect (or manifestation) of the defect. Indeed, some organisations routinely undertake root cause analysis of defects, with the aim of preventing similar problems happening in the future. It is important to realise that similar or even identical defects can have completely different root causes.

The defect is what is wrong (an incorrect calculation of interest rates), the effect is how this appears (an angry customer who is charged too much), and the root cause is why the defect came about (an incorrect understanding by the business lead of how long-term interest rates are to be calculated). So, in considering defects it is important

to distinguish between the root cause (the 'why') and the effects (the 'what happened') of a defect.

Testing and quality assurance

Testing is part of Quality Control, which itself is part of Quality Management. However, as we shall see, Quality Control encompasses MORE than testing. Quality Management comprises both Quality Control and Quality Assurance. Quality Assurance is about making sure that processes are undertaken correctly. If processes are carried out correctly, there is a greater likelihood that the end product will be better. Quality Control is about seeing whether the desired level of quality is being achieved (and if not, doing something about it). So testing is part of Quality Control (checking the quality of something – in this case, the software under test), but it does not necessarily have a part to play in courses of action if the quality does not match the desired level. And of course, checking quality can be undertaken in ways other than testing (e.g. questionnaires can be used).

Static testing and dynamic testing

Static testing is the term used for testing when the code is not exercised. This may sound strange but remember that failures often begin with a human error, namely a wrong way of thinking or an incorrect assumption (an error) when producing a document such as a specification (which will then have a defect in it). We need to test as early as possible because errors are much cheaper to fix than defects or failures (as you will see). We discussed earlier that errors are intangible, but the earlier we find something that is incorrect, the easier (and cheaper) it is to fix. That is why testing should start as early as possible (another basic principle explained in more detail later in this chapter). Static testing involves techniques such as reviews, which can be effective in preventing defects in the resulting software; for example, by removing ambiguities, omissions and faults from specification documents. This a topic in its own right and is covered in detail in **Chapter 3**. Dynamic testing is the kind that exercises the program under test with some test data, so we speak of test execution in this context. The discipline of software testing encompasses both static and dynamic testing.

Testing as a process

We have already seen that there is much more to testing than test execution. Before test execution, there is some preparatory work to do to design the tests and set them up; after test execution, there is some work needed to record the results and check whether the tests are complete. Even more important than this is deciding what we are trying to achieve with the testing and setting clear objectives for each test. A test designed to give confidence that a program functions according to its specification, for example, will be quite different from one designed to find as many defects as possible. We define a test process to ensure that we do not miss critical steps and that we do things in the right order. We will return to this important topic later, when we explain a generalised test process in detail.

Testing as a set of techniques

The final challenge is to ensure that the testing we do is effective testing. It might seem paradoxical, but a good test is one that finds a defect if there is one present. A test that finds no defect has consumed resources but added no value; a test that finds a defect has created an opportunity to improve the quality of the product. How do we design tests that find defects? We actually do two things to maximise the effectiveness of the tests. First, we use well-proven test design techniques, and a selection of the most important of these is explained in detail in **Chapter 4**. The techniques are all based on certain testing principles that have been discovered and documented over the years, and these principles are the second mechanism we use to ensure that tests are effective. Even when we cannot apply rigorous test design for some reason (such as time pressures), we can still apply the general principles to guide our testing. We turn to these next.

CHECK OF UNDERSTANDING

1. Describe static testing and dynamic testing.
2. What is debugging?
3. What other elements apart from 'test execution' are included in 'testing'?

GENERAL TESTING PRINCIPLES

Testing is a very complex activity, and the software problems described earlier highlight that it can be difficult to do well. We now describe some general testing principles that help testers, principles that have been developed over the years from a variety of sources. These are not all obvious, but their purpose is to guide testers and prevent the types of problems described previously. Testers use these principles with the test techniques described in **Chapter 4**.

Testing shows the presence, not absence, of defects

Running a test through a software system can only show that one or more defects exist. Testing cannot show that the software is error free. Consider whether the top 10 wanted criminals website was error free. There were no functional defects, yet the website failed. In this case the problem was non-functional and the absence of defects was not adequate as a criterion for release of the website into operation.

In **Chapter 2** we will discuss retesting, when a previously failed test is rerun to show that under the same conditions, the reported problem no longer exists. In this type of situation, testing can show that one particular problem no longer exists.

Although there may be other objectives, usually the main purpose of testing is to find defects. Therefore, tests should be designed to find as many defects as possible.

Exhaustive testing is impossible

If testing finds problems, then surely you would expect more testing to find additional problems, until eventually we would have found them all. We discussed exhaustive testing earlier when looking at the smartphone mapping app and concluded that for large complex systems, exhaustive testing is not possible. However, could it be possible to test small pieces of software exhaustively and only incorporate exhaustively tested code into large systems?

> Exhaustive testing – a test approach in which all possible data combinations are used. This includes implicit data combinations present in the state of the software/data at the start of testing.

Consider a small piece of software where one can enter a password, specified to contain up to three characters, with no consecutive repeating entries. Using only Western alphabetic capital letters and completing all three characters, there are 26 × 26 × 26 input permutations (not all of which will be valid). However, with a standard keyboard there are not 26 × 26 × 26 permutations, but a much higher number: 256 × 256 × 256, or 2^{24}. Even then, the number of possibilities is higher. What happens if three characters are entered, and the 'delete last character' right arrow key removes the last two? Are special key combinations accepted, or do they cause system actions (Ctrl + P, for example)? What about entering a character, and waiting 20 minutes before entering the other two characters? It may be the same combination of keystrokes, but the circumstances are different. We can also include the situation where the 20-minute break occurs over the change-of-day interval. It is not possible to say whether there are any defects until all possible input combinations have been tried.

Even in this small example, there are many, many possible data combinations to attempt. The number of possible combinations using a smartphone might be significantly less, but it is still large enough to be impractical to use all of them.

Unless the application under test (AUT) has an extremely simple logical structure and limited input, it is not possible to test all possible combinations of data input and circumstances. For this reason, risk and priorities are used to concentrate on the most important aspects to test. Both 'risk' and 'priorities' are covered later in more detail. Their use is important to ensure that the most important parts are tested.

Early testing saves time and money

When discussing why software fails, we briefly mentioned the idea of early testing. This principle is important because, as a proposed deployment date approaches, time pressure can increase dramatically. There is a real danger that testing will be squeezed, and this is bad news if the only testing we are doing is after all the development has been completed. The earlier the testing activity is started, the longer the elapsed time available. Testers do not have to wait until software is available to test.

Work products are created throughout the Software Development Life Cycle (SDLC), and we talk about these different work products later in this chapter. As soon as these are ready, we can test them. In **Chapter 2**, we will see that requirement documents are the basis for acceptance testing, so the creation of acceptance tests can begin as soon as requirement documents are available. As we create these tests, they will highlight the contents of the requirements. Are individual requirements testable? Can we find ambiguous or missing requirements?

Many problems in software systems can be traced back to missing or incorrect requirements. We will look at this in more detail when we discuss reviews in **Chapter 3**. The use of reviews can break the 'error–defect–failure' cycle. In early testing, we are trying to find errors and defects before they are passed to the next stage of the development process. Early testing techniques are attempting to show that what is produced as a system specification, for example, accurately reflects that which is in the requirement documents. Ed Kit discusses identifying and eliminating defects at the part of the SDLC in which they are introduced.[1] If an error/defect is introduced in the coding activity, it is preferable to detect and correct it at this stage. If a problem is not corrected at the stage in which it is introduced, this leads to what Kit calls 'errors of migration'. The result is rework. We need to rework not just the part where the mistake was made, but each subsequent part where the error has been replicated. A defect found at acceptance testing where the original mistake was in the requirements will require several work products to be reworked, and subsequently to be retested.

Studies have been done on the cost impacts of errors at the different development stages. While it is difficult to put figures on the relative costs of finding defects at different levels in the SDLC, Table 1.1 does concentrate the mind!

Table 1.1 Comparative cost to correct errors

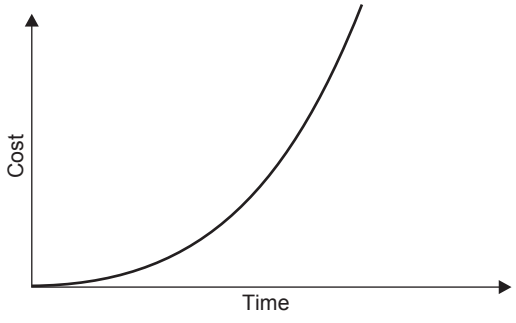
Stage error is found	Comparative cost
Requirements	$1
Coding	$10
Program testing	$100
System testing	$1,000
User acceptance testing	$10,000
Live running	$100,000

This is known as the cost escalation model.

What is undoubtedly true is that the graph of the relative cost of early and late identification/correction of defects rises very steeply (Figure 1.3).

1 Kit, E. (1995) Software Testing in the Real World. Reading, MA: Addison-Wesley.

Figure 1.3 Effect of identification time on cost of errors



The earlier a problem (defect) is found, the less it costs to fix.

The objectives of various stages of testing can be different. For example, in the review processes, we may focus on whether the documents are consistent and no defects have been introduced when the documents were produced. Other stages of testing can have other objectives. The important point is that testing has defined objectives.

One of the drivers behind the push to Agile development methodologies is to enable testing to be incorporated throughout the software build process. This is nothing more than the 'early testing' principle.

Defects cluster together

Problems do occur in software. It is a fact. Once testing has identified (most of) the defects in a particular application, it is at first surprising that the spread of defects is not uniform. In a large application, it is often a small number of modules that exhibit the majority of the problems. This can be for a variety of reasons, some of which are:

- system complexity;
- volatile code;
- the effects of change on change;
- development staff experience;
- development staff inexperience.

This is the application of the Pareto principle to software testing: approximately 80 per cent of the problems are found in about 20 per cent of the modules. It is useful if testing activity reflects this spread of defects, and targets areas of the application under test where a high proportion of defects can be found. However, it must be remembered that testing should not concentrate exclusively on these parts. There may be fewer defects in the remaining code, but testers still need to search diligently for them.

Be aware of the pesticide paradox

Running the same set of tests continually will not continue to find new defects. Developers will soon know that the test team always tests the boundaries of conditions, for example, so they will learn to test these conditions themselves before the software is delivered. This does not make defects elsewhere in the code less likely, so continuing to use the same test set will result in decreasing the effectiveness of the tests. Using other techniques will find different defects.

For example, a small change to software could be specifically tested and an additional set of tests performed, aimed at showing that no additional problems have been introduced (this is known as regression testing). However, the software may fail in production because the regression tests are no longer relevant to the requirements of the system or the test objectives. Any regression test set needs to change to reflect business needs, and what are now seen as the most important risks. Regression testing will be covered in more detail in **Chapter 2**.

Testing is context dependent

Different testing is necessary in different circumstances. A website where information can merely be viewed will be tested in a different way to an ecommerce site, where goods can be bought using credit/debit cards. We need to test an air traffic control system with more rigour than an application for calculating the length of a mortgage.

Risk can be a large factor in determining the type of testing that is needed. The higher the possibility of losses, the more we need to invest in testing the software before it is implemented. A fuller discussion of risk is given in **Chapter 5**.

For an ecommerce site, we should concentrate on security aspects. Is it possible to bypass the use of passwords? Can 'payment' be made with an invalid credit card, by entering excessive data into the card number field? Security testing is an example of a specialist area, not appropriate for all applications. Such types of testing may require specialist staff and software tools. Test tools are covered in more detail in **Chapter 6**.

Absence-of-errors is a fallacy

Software with no known errors is not necessarily ready to be shipped. Does the application under test match up to the users' expectations of it? The fact that no defects are outstanding is not a good reason to ship the software.

Before dynamic testing has begun, there are no defects reported against the code delivered. Does this mean that software that has not been tested (but has no outstanding defects against it) can be shipped? We think not.

CHECK OF UNDERSTANDING

1. Why is 'zero defects' an insufficient guide to software quality?
2. Give three reasons why defect clustering may exist.
3. Briefly justify the idea of early testing.

TEST PROCESS

We previously determined that testing is a process, discussed above. It would be easy to think that testing is thus always the same. However, this is not true.

Test process in context

Variations between organisations, and indeed projects within the same organisation, can have an influence on how testing is carried out, and on the specific test process that is used. Specific matters, or contextual factors, that will affect the test process can be many and varied, so please do not assume that the factors listed in the syllabus are the **only** factors. Many of these are covered in more detail in later chapters of this book. We give those in the syllabus here, and remember, you could be examined on these:

- the Software Development Life Cycle and project methodologies that are in use;
- test levels and test types being considered;
- product and project risks (if there is potential loss of life, you will generally 'test more');
- the business domain (online games are possibly tested in different ways to a billing system in a utility company);
- operational constraints, which could include the following, but also other matters:
  - budgets and other resources (including staffing levels);
  - timescale, and whether there are time-to-market constraints;
  - complexity (however this is measured);
  - any contractual or regulation requirements; for example both motor manufacturing and the pharmaceutical industries have special industry-wide regulations.
- any policies and practices that are specific to the organisation;
- required standards, both internal and external.

Test activities and tasks

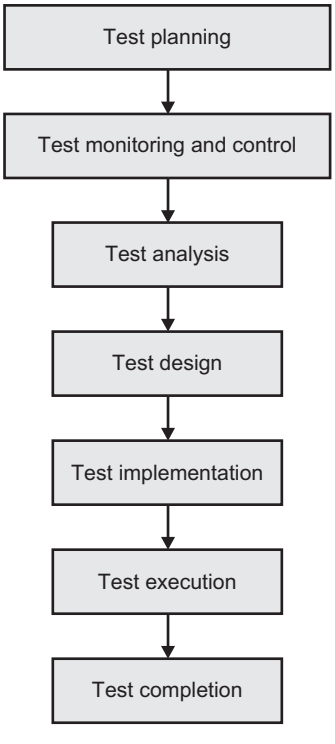
The most visible part of testing is running one or more tests: test execution. We also have to prepare for running tests, analyse the tests that have been run and see whether testing is complete. Both planning and analysing are very necessary activities that

enhance and amplify the benefits of the test execution itself. It is no good testing without deciding how, when and what to test. Planning is also required for the less formal test approaches such as exploratory testing, covered in more detail in **Chapter 4**. We are describing a generalised test process – as was said earlier, this will not be the same on every project within an organisation, nor between different organisations. Not all of the activity groups that are described will be recognised as separate entities in some projects, or within certain organisations.

So, there are the following activity groups in the test process that we will describe in more detail (Figure 1.4):

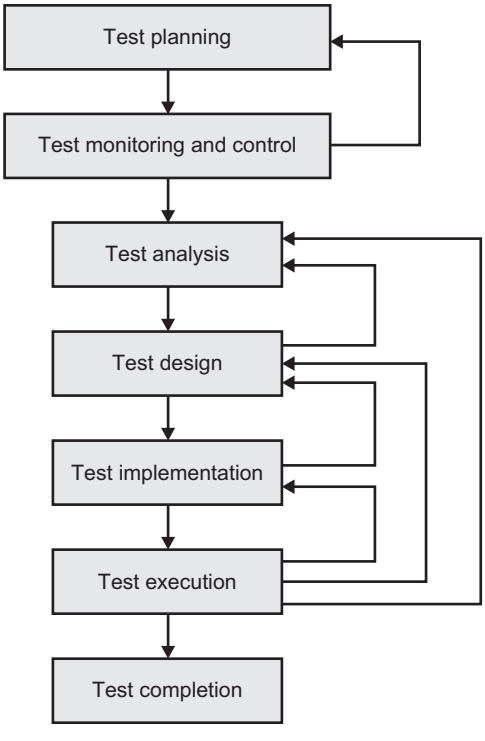
1. test planning;
2. test monitoring and control;
3. test analysis;
4. test design;
5. test implementation;
6. test execution;
7. test completion.

Figure 1.4 A generalised test process



Although the main activity groups are in a broad sequence, they are not undertaken in a rigid way. The first two groups are overarching, and the other groups are both planned and monitored, and what has been undertaken may have a bearing upon other actions, some of which have been completed. An earlier activity group may need to be revisited. A defect found in test execution can sometimes be resolved by adding functionality that was originally not present (either missing, or the new functionality is needed to make the other part correct). The new features themselves have to be tested, so even though implementation and execution are in progress, the 'earlier' activity groups of test analysis and test design have to be performed for the new features (Figure 1.5). The activity groups may overlap or be performed concurrently. It could be that test design is being undertaken for one test level at the same time as test execution is underway for another test level on the same project (test execution for system testing and test design for acceptance testing, for example). Please note, not all possible interactions between later and earlier groups have been included, and all of the latter five stages have interaction between **both** test planning and test monitoring and control

Figure 1.5 Iteration of activities



We sometimes need to do two or more of the main activities in parallel. Time pressure can mean that we begin test execution before all tests have been designed.

Test planning

Planning is determining what is going to be tested, and how this will be achieved. It is where we draw a map; how activities will be done; and who will do them. Test planning is also where we define the test completion criteria. Completion criteria are how we know when testing is finished. It is where the objectives of testing are set, and where the approach for meeting test objectives (within any context-based limitations) are set. A test plan is created and schedules are drawn up to enable any internal or external deadlines to be met. Plans and schedules may be amended, depending upon how other activities are progressing – amendments take place as a result of monitoring and control activities. Planning is described in more detail in **Chapter 5**.

Test monitoring and control

Test monitoring and control go together. Monitoring is concerned with seeing if what has been achieved is what was expected to be done at this point in time, whereas control is taking any necessary action to meet the original or revised objectives as given in the test plan. Monitoring and control are supported by referring to the exit criteria or completion criteria for different stages of testing – this is sometimes referred to as the definition of 'done' in some project life cycles. Looking at exit criteria for test execution for a stage of testing may include:

- checking test results to see if the required test coverage has been achieved;
- determining the component or system quality, by looking at both test results and test logs;
- seeing if more tests are required (for example, if there are not enough tests to reach the level of risk coverage that is required).

Progress against the (original or revised) test plan is communicated to the necessary people (project sponsor, user stakeholders, the development team) in test progress reports. These will usually detail any actions being put in place to enable some milestones to be met earlier than would otherwise be the case or help to inform any decisions that stakeholders need to make. Test monitoring and control are covered more fully in **Chapter 5**.

Test analysis

Test analysis examines the test basis (which is usually written but not always confined to a single document), to identify what to test. Testable parts are identified, and test conditions drawn up. Major activities in the test analysis activity group are as follows

- examining the test basis relevant for the testing to be carried out at this time:
  - requirements documents, functional requirements, business requirements, system requirements or other items (use cases, user stories) that detail both functional **and** non-functional component or system behaviour;

  - design or implementation detail (system or software architecture, entity-relationship diagrams, interface specifications) that define the component or system structure;
  - implementation information for the component or system, including code, database metadata and queries;
  - risk analysis reports, which may consider both functional and non-functional aspects, and the structure of the software to be tested.
- looking at the test basis looking for defects of various types:
  - ambiguities;
  - things that have been missed out;
  - inconsistencies;
  - inaccuracies;
  - contradictions;
  - unnecessary or superfluous statements.
- identifying features and feature sets to be tested;
- identifying and prioritising test conditions for each feature or set of features (based on the test basis). Prioritisation is based on functional, non-functional and structural factors, other matters (both business and technical) and the levels of risk;
- capturing traceability in both directions ('bi-directional traceability') between the test basis and test conditions.

It can be both appropriate and very helpful to use some of the test techniques that are detailed in **Chapter 4** in the test analysis activity. Test techniques (black box, white box and experience-based techniques) can assist to define more precise and accurate test conditions and include important but less obvious test conditions.

Sometimes, test conditions produced as part of test analysis are used as test objectives in test charters. Test charters can be important in exploratory testing, discussed as part of experience-based test techniques in **Chapter 4**.

The test analysis activity can identify defects in the test basis. This can be especially important when there is no additional review process in place, or where the test process is closely aligned to the review process. Test design activity by its very nature looks at the test basis (including requirements), so it is appropriate to see whether requirements are consistent, clearly expressed and not missing anything. The work of test analysis can also rightly ask whether formal requirements documents have included customer, user and other stakeholder needs. Some development methodologies involve creating test conditions and test cases prior to any coding taking place. Examples of such methodologies include behaviour driven development (BDD) and acceptance test driven development (ATDD).

Test design

Test analysis answers the question 'What to test?', while test design answers the question 'How to test?' It is during the test design activity that test conditions are used to create test cases. This may also use test techniques, and the very process of creating test cases can identify defects. As we discussed earlier, a test case not only describes what to do to see if the test condition is correct but also what the expected result should be. Creating two or more test cases for a test condition can identify actions where the expected result is not clear – in the example we used earlier, are there different error messages if the user name is incorrect and the password is incorrect? If the requirements, specification or other documents are not clear, there is potential for misunderstandings!

Activities in the test design activity group can be summarised as follows:

- using test conditions to create test cases, and prioritising both test cases and sets of test cases;
- identifying any test data to be used with the test cases;
- designing the test environment if necessary, and identifying any other items that are needed for testing to take place (infrastructure and any tools required);
- detailing bi-directional traceability between test basis, test conditions, test cases and test procedures – building upon the traceability that was detailed above under test analysis.

Test implementation

We saw that test design asks 'How to test?' Test implementation asks 'Do we have everything in place to run the tests?' It is the link between test design and running the test, or test execution. We have the tests ready; how can we run them? Test implementation is not always a separate activity; it can be combined with test design, or, in exploratory testing and some other kinds of experienced-based testing, test design and test implementation may take place and be documented as part of the running of tests. In exploratory testing, tests are designed, implemented and executed simultaneously.

As we get ready for test execution, the test implementation main activities are as follows:

- creating and prioritising test procedures and possibly creating automated test scripts;
- creating test suites from the test procedures and from automated test scripts if there are any;
- ordering test suites in a test execution schedule, so that it makes efficient use of resources (it is often the case that a test execution schedule will **create** an order, **amend** the order and then **delete** it – all done to run efficiently);

- building the test environment with anything extra in place (test harnesses, simulators, dummy third-party interfaces, service virtualisation and any other infrastructure items). The test environment has then to be checked, to ensure that all is ready to start testing;
- preparing test data and checking that it has been loaded into the test environment;
- checking and updating the bi-directional traceability between the test basis, test conditions, test cases, test procedures and now to include test suites.

Test execution

Test execution involves running tests, and is where (what are often seen as) the most visible test activities are undertaken. Test suites are run as detailed in the test execution schedule (both of which were created as part of test implementation). Execution includes the following activities:

- Recording the identification and version of what is being tested: test items or test object, test tool(s) and any testware that are in use. This information is important if any defects are to be raised.
- Running tests, either manually or using test execution tools.
- Comparing actual and expected results (this may be done by any test execution tool(s) being used).
- Looking at instances where the actual result and the expected result differ to determine the possible cause. This may be a defect in the software under test, but could also be that the expected results were incorrect, or the test data was not quite correct.
- Reporting defects based on the failures that were found in the testing.
- Recording the result of each test (pass, fail, blocked).
- Repeating tests where the software has been corrected, the test data has been changed or as part of regression testing).
- Confirming or amending the bi-directional traceability between the test basis, the test conditions, test cases, test procedures and expected results.

Test completion

Testing at this stage has finished, and test completion activities collect data so that lessons can be learned, testware reused in future projects and so on. These activities may occur at the time that the software system is released, but could also be at other significant project milestones – the project is completed (or even cancelled), a test level is completed or an Agile project iteration is finished (where test completion activities may be part of the iteration retrospective meeting). The key here is to make sure that information is not lost (including the experiences of those involved). Activities in the test completion group are summarised below:

- Checking that defect reports are all closed as necessary. This may result in the raising of change requests or creating product backlog items for any that remain unresolved when test execution activities have ended.
- Creating a test summary report, to communicate the results of the testing activities. This is usually for the benefit of the project stakeholders.
- Closing down and saving ('archiving') the test environment, any test data, the test infrastructure and other testware. These may need to be reused at a later date.
- Handing over the testware to those who will maintain the software in the future or any other projects or other stakeholders to whom it could be beneficial.
- Analysing lessons learned from testing activities that have been completed, which will hopefully drive changes in future iterations, releases and projects.
- Using the information that has been gathered to make the testing process better – to improve test process maturity.

Agile methodologies and a generalised test process

Thus far we have concentrated on 'traditional' methodologies when discussing the test process. We now want to focus on Agile methodologies.

The use of Agile methodologies and the relationship with a test process is not a syllabus topic for the Foundation Certificate, but in the following discussion our understanding of both a generalised test process and Agile methodologies will grow.

We use the term 'Agile methodologies' because there is not a single variant or 'flavour' of Agile. Any distinctions between these are unimportant here, but the principles are important. Agile Software Development Life Cycles aim to have frequent deliveries of software, where each iteration or sprint will build on any previously made available. The aim is to have software that could be implemented into the production environment at the end of each sprint – although a delivery into production might not happen as frequently as this.

From this brief introduction to Agile, it follows that an Agile project has a beginning and an end, but the processes that go between these stages can be repeated many times. Agile projects can be successfully progressing over the course of many months, or even years, with a continuous stream of (same length) iterations producing production-quality software, typically every two, three or four weeks. In terms of the test process, there is a part of the test planning stage that takes place at the start of the project, and the test completion activities take place at the end of the project. However, some test planning, and all of the middle five stages of the test process we have described above are present in each and every sprint.

Some test planning activities take place at the beginning of the project. This typically includes resourcing activities, some outline planning on the length of the project, and an initial attempt at ordering the features to be implemented (although this could change significantly as the project progresses, sprint by sprint). Infrastructure planning, together with the identification and provision of any specific testing tools is usually

undertaken at this stage, together with a clear understanding within the whole team of what is 'done' (i.e. a definition of 'done').

At the start of each sprint, planning activity takes place, to determine items to include in the sprint. This is a whole-team activity based on reaching a consensus of how long each potential deliverable will take. As sprint follows sprint, so the accuracy of estimation increases. The sprint is a fixed length, so throughout the duration of the development items could be added or removed to ensure that, at the conclusion, there is tested code 'on the shelf' and available for implementation as required.

Other activities of the generalised test process we described are undertaken in each sprint. A daily stand-up meeting should result in a short feedback loop, to enable any adjustments to take place and items that prevent progress to be resolved. The whole time for development and testing is limited, so the preparation for testing and the testing itself have to be undertaken in parallel. Towards the end of the sprint, it is possible that most or all of the team are involved in testing, with a focus of delivering all that was agreed at the time of the sprint planning meeting.

Automated testing tools are frequently used in Agile projects. Tests can be very low level, and a tester on such a project can provide very useful input to developers in defining tests to be used to identify specific conditions. The earlier in a sprint that this is done, the more advantages can be gained within that sprint.

The proposed deliverables are sometimes used in conjunction with the definition of 'done' to enable a burn-down chart to be drawn up. This enables a track to be kept of progress for all to see – in itself usually a motivating factor. As sprint follows sprint, the subject of regression testing previously delivered, working software becomes more important. Part of the test analysis and test design will involve selecting regression tests that are appropriate for the current sprint. Tests that previously worked might now (correctly) not pass because the new sprint has changed the intention of previously delivered software. The conclusion of the sprint is often a sprint review meeting, which can include a demonstration to user representatives and/or the project sponsor.

For development using Agile methodologies, the final stage of our test process – 'test completion activities' – is scheduled after the end of the last sprint. This should not be done before, because testing collateral that was used in the last-but-three sprint might no longer be appropriate at the conclusion of the final sprint. As we discussed earlier, even the regression tests might have changed, or a more suitable set of regression tests identified.

Further information about Agile methodologies is given in **Chapter 2**.

Test work products

Each of the activity groups we have discussed has one or more work products that are typically produced as part of that set of activities. However, just as there are major variations in the way particular organisations implement the test process, so there can be variations in both the work products that are produced, and in some cases even the names of those work products. This section follows the test process that we have

outlined above, and work products are given for each of the seven activity groups that we described in a generalised test process.

Many of the work products that are described here can be captured and managed using test management and defect management tools (these two tool types are both described in more detail in **Chapter 6**). Work products used in test process activity groups can often be easily attributed to the appropriate activity group or when considering the preceding or following activity group, so even though 'Test work products' can be a topic in the examination, this does not mean that you have to learn the lists that are given! An example of an examination-type question relating to this area of study is given at the end of this chapter.

There are some work products that are typically created in two or more test process activity groups. Examples of this are defect reports (most usually in test analysis, test design, test implementation and test execution) and test reports (test progress reports and test summary reports in test monitoring and control, test completion reports in both test monitoring and control, and test completion).

Test planning work products

Test planning activities usually produce plans and schedules. There can be several test plans for a project: component testing test plan, integration testing test plan and so on. Test plans contain information about the test basis and the exit criteria (or definition of 'done'), so that we know in advance when we can say that testing is complete. Other work products will be related to the test basis by traceability information.

Test monitoring and control work products

Work products produced from the test monitoring and control activities include various types of test reports. Some of these are periodic (a test progress report every three weeks, perhaps), while others are at specific completed milestones for the project (test summary reports). Not all reports are for the same target audience, so any reports need to be audience specific in the level of detail. Reports for senior stakeholders may just give the number of tests that have passed, have failed and cannot yet be run, for example, whereas reports for the wider development team may include sub-system-specific information about the tests that have passed and failed.

Test monitoring and control work products also need to highlight project management concerns including task completion, the use of resources and the amount of effort that has been expended. Where necessary, there may be choices highlighted with possible actions that can be taken to still achieve the original timescales, even though at the present time we are behind schedule.

Test analysis work products

The following are work products that the test analysis group of activities is expected to produce:

- Defined and prioritised test conditions (ideally each with bi-directional traceability to the specific part(s) of the test basis that is covered).
- For exploratory testing, test charters may be created.
- Test analysis may result in the discovery and reporting of defects in the test basis.

Test design work products

The key work products that come from test design activities are test cases and groups of test cases that relate to the test conditions created by test analysis. It is extremely useful if these are bi-directionally traceable to the test conditions they relate to. Test cases can be high-level test cases (without specific values, or concrete values to be used for input values) or low-level test cases (with detailed input values, and **specific expected results**). High-level test cases can be reused in different test cycles, although they are not as easy to use by individuals who are not familiar with the software being tested.

Other work products that are produced or amended as part of test design include amendments to the test conditions created in test analysis, the design and/or identification of any test data that is needed, the definition and design of the test environment, and the identification of any infrastructure or tools that may be required. However, although these last mentioned are done, the amount that is documented can vary.

Test implementation work products

Test implementation activities have some work products that are more easily identifiable than others. The first three below are very recognisable, but the others can be also created at this stage:

- Test procedures, and the sequencing of these.
- Test suites, being comprised of two or more test procedures.
- A test execution schedule.
- In some cases, test implementation activities create work products using or used by tools:
  - service virtualisation;
  - automated test scripts.
- The creation and verification of test data.
- Creation and verification of the test environment.
- Further refinement of the test conditions that were produced as part of test analysis.

When we have specific test data, this can be used to turn high-level test cases into low-level test cases, and to these are added expected results. In some instances, expected

results can be derived from the test data using a test oracle – you feed the data in and get the expected results as the answer.

Once test implementation is complete, it can be possible to see the level of coverage by written test cases for parts or all of the test basis. This is because at each stage, we have built-in, bi-directional traceability. It may be possible to see how many test procedures have been written that cover a specific requirement, or area of functionality.

Test execution work products

The three key work products created by test execution activities are given below:

- The status of individual tests (passed, failed, skipped, ready to run, blocked, etc.).
- Defect reports as a result of test execution.
- Details of what was involved in the testing (test item(s), test objects, testware and test tools). This includes the version identifier of each of these items – so that if necessary, the exact test can be replicated at a future time.

When testing is complete, if there is bi-directional traceability between the test basis, test conditions, test cases, test procedures and test suites, it is possible to work backwards and say which requirements have failed tests against them, where the impact of defects touches the business areas and so on. This will enable checking that the pre-determined coverage level has been met, or not, and help reporting in ways that business stakeholders understand.

Test completion work products

When testing is complete, the following can be created from the test completion activities:

- test summary reports, which could be a 'test completion report';
- a list of improvements for future work (the next iteration, or next project);
- change requests or product backlog items;
- finalised testware (and sometimes, the test environment) for future usage.

Traceability between the test basis and test work products

Throughout the descriptions of the test process groups of activities, and the work products that are created by these activity groups, there has been a sub-story of bi-directional traceability. Primarily, this enables the evaluation of test coverage: how does this one failed test reflect back to requirements or business goals? There are several great advantages to having full traceability:

- assessing the impact of changes (if one requirement changes, how many test conditions, test cases, test procedures and test suites may be affected);
- making testing auditable;

- enabling IT governance criteria to be met;
- improving the understandability of test progress reports and test summary reports to reflect passed, failed and blocked tests back to requirements or other aspects of the test basis;
- relating testing to stakeholders in terms that they can understand (it is not 'three failed tests' but 'three tests failed, which means that creating a customer is not possible');
- enabling the assessment of product quality, process capability and project progress against the goals of the business.

CHECK OF UNDERSTANDING

1. What are the activity groups in the generalised test process described (in the correct sequence)?
2. Give advantages of maintaining traceability between the test basis and test work products (including test cases).
3. When should the expected outcome of a test be defined?
4. Give three work products that are created during the test execution group of activities.

THE PSYCHOLOGY OF TESTING

A variety of different people may be involved in the total testing effort, and they may be drawn from a broad set of backgrounds. Some will be developers, some professional testers and some will be specialists, such as those with performance testing skills, while others may be users drafted in to assist with acceptance testing. Whoever is involved in testing needs at least some understanding of the skills and techniques of testing to make an effective contribution to the overall testing effort.

Testing can be more effective if it is not undertaken by the individual(s) who wrote the code, for the simple reason that the creator of anything (whether it is software or a work of art) has a special relationship with the created object. The nature of that relationship is such that flaws in the created object are rendered invisible to the creator. For that reason, it is important that someone other than the creator should test the object. Of course, we do want the developer who builds a component or system to debug it, and even to attempt to test it, but we accept that testing done by that individual cannot be assumed to be complete. Developers can test their own code, but it requires a mindset change, from that of a developer (to prove it works) to that of a tester (trying to show that it does not work).

Testers and developers think in different ways. However, although we know that testers should be involved from the beginning, it is not always good to get testers involved

in code execution at an early stage; there are advantages and disadvantages. Getting developers to test their own code has advantages (as soon as problems are discovered, they can be fixed, without the need for extensive error logs), but also difficulties (it is hard to find your own mistakes – the so-called 'confirmation bias'). People and projects have objectives, and we all modify actions to blend in with the goals. If a developer has a goal of producing acceptable software by certain dates, then any testing is aimed towards that goal.

If a defect is found in software, the software author may see this as criticism. Testers need to use tact and diplomacy when raising defect reports. Defect reports need to be raised against the software, not against the individual who made the mistake. The mistake may be in the code written, or in one of the documents on which the code is based (requirement documents or system specification). When we raise defects in a constructive way, bad feeling can be avoided.

We all need to focus on good communication, and work on team building. Testers and developers are not opposed but working together, with the joint target of better-quality systems. Communication needs to be objective, and expressed in impersonal ways:

- The aim is to work together rather than be confrontational. Keep the focus on delivering a quality product.
- Results should be presented in a non-personal way. The work product may be wrong, so say this in a non-personal way.
- Attempt to understand how others feel; it is possible to discuss problems and still leave all parties feeling positive.
- At the end of discussions, confirm that you have both understood and been understood. 'So, am I right in saying that you will aim to deliver with all the agreed priority fixes on Friday this week by 12.00?'

As testers and developers, one of our goals is better-quality systems delivered in a timely manner. Good communication between testers and the development teams is one way that this goal can be reached.

CHECK OF UNDERSTANDING

1. Describe ways in which testers and developers think differently.
2. Contrast the advantages and disadvantages of developers testing their own code.
3. Suggest three ways that confrontation can be avoided.

CODE OF ETHICS

It should be noted that this section is **not** examinable but is retained (being in earlier versions of the syllabus, but not the current one), as it is a useful topic for the tester and aspiring tester to be aware of.

We will now look at how testers should behave as professionals in the workplace, a code of ethics, before we move onto the more detailed coverage of topics in the following chapters. Testers can have access to confidential and/or privileged information, and they are to treat any information with care and attention, and act responsibly when dealing with the owner(s) of this information, employers and the wider public interest. Of course, anyone can test software, so the declaration of this code of ethics applies to those who have achieved software testing certification. The code of ethics applies to the following areas:

- Public – certified software testers shall consider the wider public interest in their actions.
- Client and employer – certified software testers shall act in the best interests of their client and employer (being consistent with the wider public interest).
- Product – certified software testers shall ensure that the deliverables they provide (for any products and systems they work on) meet the highest professional standards possible.
- Judgement – certified software testers shall maintain integrity and independence in their professional judgement.
- Management – certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.
- Profession – certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.
- Colleagues – certified software testers shall be fair to, and supportive of, their colleagues and promote cooperation with software developers.
- Self – certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

The code of ethics is far-reaching in its aims, and a quick review of the eight points reveals interaction with specific areas of the syllabus. The implementation of this code of ethics is expanded on in all chapters of this book, and perhaps is the reason for the whole book itself.

SUMMARY

In this chapter, we have looked at key ideas that are used in testing and introduced some terminology. We examined some of the types of software problems that can occur, and why the blanket explanation of 'insufficient testing' is unhelpful. The problems encountered then led us through some questions about the nature of testing, why errors and mistakes are made, and how these can be identified and eliminated. Individual examples enabled us to look at what testing can achieve, and the view that testing does not improve software quality but provides information about that quality.

We have examined both general testing principles and a standard template for testing: a generalised test process. These are useful and can be effective in identifying the types of

problems we considered at the start of the chapter. The chapter finished by examining how developers and testers think, and how testers should behave by adhering to a code of ethics.

This chapter is an introduction to testing, and to themes that are developed later in the book. It is a chapter in its own right, but also points to information that will come later. A rereading of this chapter when you have worked through the rest of the book will place all the main topics into context.

Example examination questions with answers

E1. K1 question
What is a test condition?

a. A set of test data written to exercise one or more logic paths through the software under test.
b. An aspect of the test basis appropriate to achieve specific test objectives.
c. The body of knowledge used as the foundation for test analysis and design.
d. The pre-determined goals that will enable a decision to be made about whether testing is complete.

E2. K2 question
Which of the following definition pairs for testing and debugging is correct?

a. Debugging can show failures in the software; testing is investigating the causes of any failures and performing corrections.
b. Debugging is investigating the causes of software failures and undertaking corrective action; testing attempts to uncover problems by executing the software.
c. Debugging is always undertaken by developers; testing can be performed by development or testing personnel.
d. Debugging checks that software fixes have been resolved; testing is looking for any unintended consequences of a software fix.

E3. K1 question
Which of the following are aids to good communication within the development team?

i. Try to understand how the other person feels.
ii. Communicate personal feelings, concentrating on individuals.
iii. Confirm the other person has understood what you have said and vice versa.
iv. Emphasise the common goal of better quality.
v. Each discussion is a battle to be won.

a. i, ii and iii aid good communication.
b. iii, iv and v aid good communication.
c. i, iii and iv aid good communication.
d. ii, iii and iv aid good communication.

E4. K2 question
Which of the following illustrates one of the testing principles?

a. No unresolved defects does not mean the software will be successful.
b. The more you test, the more defects will be found.
c. All software can be tested in the same way using the same test techniques.
d. Defects are usually found evenly distributed throughout the software under test.

E5. K2 question
Which of the following activities are part of the test implementation activity group, and which part of test execution?

i. Developing and prioritising test procedures, creating automated test scripts.
ii. Comparing actual and expected results.
iii. Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures and test results.
iv. Preparing test data and ensuring it is properly loaded in the test environment.
v. Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures and test suites.

a. i, ii and iii are part of test implementation, iv and v are part of test execution.
b. i, iii and v are part of test implementation, ii and iv are part of test execution.
c. i, ii and iv are part of test implementation, iii and v are part of test execution.
d. i iv and v are part of test implementation, ii and iii are part of test execution.

E6. K2 question
Which of the following ways of thinking, broadly apply to developers and which to testers?

i. Curiosity, professional pessimism and a critical eye.
ii. Interested in designing and building solutions.
iii. Can be subject to the confirmation bias.
iv. Has good attention to detail.
v. Contemplates what might go wrong.

a. ii and iii: developers; i, iv and v: testers.
b. i and ii: developers; iii, iv and v: testers.
c. iv and v: developers; i, ii and iii: testers.
d. iii and v: developers; i, ii and iv: testers.

E7. K2 question
Which of the following is a recognised reason for testing to be carried out?

a. Using testers in the review of requirements will verify that the software is fit for purpose.
b. Evidence suggests that between 25% and 45% of the project costs should be used in the testing process.
c. The detection and removal of defects increases the likelihood that the software meets stakeholder needs.
d. Testing ensures that there are no residual defects in the software under test.

E8. K2 question
Problems persist with the online customer interface for a utilities company, after the introduction of smart meters in customer homes. Which *one* of the following is a root cause of a defect rather than an effect of a defect?

a. Consumption usage for some days is shown as zero, while that for other days is abnormally high.
b. Some chosen menu options intermittently display the 'system busy' icon, but never the correct information at the time it is selected.
c. Customer consumption data is only displayed for up to three days ago, with year-to-date and month-to-date excluding the last three days.
d. Systems architects did not anticipate the amount of web traffic the introduction of smart meters would generate.

E9. K2 question
Which *two* of the following work products are created during the test implementation activity?

i. Documentation about which test item(s), test object(s), test tools and testware were involved in the testing.
ii. Test execution schedule.
iii. Test cases.
iv. Documentation about the **status** (e.g. 'pass', 'fail', 'not run' etc.) of individual test cases or procedures.
v. Test procedures and their sequencing.

a. iii and v.
b. ii and v.
c. i and iii.
d. ii and iv.

Answers to questions in the chapter

SA1. The correct answer is c.
SA2. The correct answer is b.
SA3. The correct answer is d.

Answers to example examination questions

E1. The correct answer is b.

a. is the definition of a test case.
c. is the description of a test basis.
d. is a broad description of test exit criteria.

E2. The correct answer is b.

a. the two parts are 'switched'; what is described as 'debugging' is in fact 'testing', and vice versa.
c. is not true. The syllabus states that in some life cycles, testers may be involved in debugging.
d. indicates that debugging is involved in retesting software after fixes. This is a description of 'retesting', a testing activity. The description given to 'testing' is that of regression testing; testing involves more than this.

E3. The correct answer is c.
Choices ii and v are factors that will **not** help good communication, but will cause mistrust, antagonism and stress in the team. These are ruled out. Option c is the answer that has the other choices. A quick check sees that these are all matters that will encourage openness and trust – and are therefore correct choices.

E4. The correct answer is a.

b. this option has **some** truth to it, but it is not one of the testing principles. However, if there are no defects in the code, no amount of testing will find defects.
c. is not true. It is in direct contradiction to the 'testing is context dependent' principle.
d. is again not true. This is usually found **not** to be the case, being the opposite of the 'defect clustering' principle.

E5. The correct answer is d.
Two of the choices are very similar, choices iii and v. The differences here is **test results** (choice iii) and **test suites** (choice v). This points to choice iii being part of test execution and choice v being part of test implementation. Option d is the only one that has these assigned in this way.

E6. The correct answer is a.
This question is not implying that all developers think in one way and all testers in another, nor that individual developers cannot exhibit 'think as testers' characteristics. The choices provided are fairly straightforward, with the exception of iv (has good attention to detail). This last choice is given as an attribute of the way a tester thinks, but is not exclusive to testers! Choices i and iii are more aligned to testers, whereas choices ii and v relate to developers. This points to option a being the correct answer.

E7. The correct answer is c.
Reviewing **requirements** can never verify that **software** is fit for purpose (option a). Just because evidence **may** suggest that a proportion of project cost should be spent on testing is not a reason to perform testing. This rules out option b. Option d states that testing can find all defects, which means that we can show that there are no remaining defects – contrary to one of the testing principles about testing only finding defects; it cannot show that are **no defects**. This leaves option c, which is the correct answer.

E8. The correct answer is d.
A root cause is **why** something has happened, as opposed to what has happened. Options a, b and c describe unusual events (which may or may not be defects but are certainly irritating for customers). Option b could be a system overload problem – this could be as a result of a higher than expected amount of web traffic. This root cause is described in option d, the correct answer.

E9. The correct answer is b.
We will consider each of the work products in turn:

i. Documentation about which test item(s), test object(s), test tools and testware were involved in the testing – **test execution**.
ii. Test execution schedule – **test implementation**.
iii. Test cases – **test execution**.
iv. Documentation about the **status** (e.g. 'pass', 'fail', 'not run' etc.) of individual test cases or procedures – **test implementation**.
v. Test procedures and their sequencing – **test execution**.

Option b gives the correct choices.

2 LIFE CYCLES

Angelina Samaroo

INTRODUCTION

In the previous chapter, we looked at testing as a concept – what it is and why we should do it. In this chapter, we will look at testing as part of the overall software development process. Clearly, testing does not take place in isolation; there must be a product first.

We will refer to work products and products. A work product is an intermediate deliverable required to create the final product. Work products can be documentation or code. The code and associated documentation will become the product when the system is declared ready for release. In software development, work products are generally created in a series of defined stages, from capturing a customer requirement, to creating the system, to delivering the system. These stages are usually shown as steps within a Software Development Life Cycle.

In this chapter, we will look at two life cycle models – sequential and iterative. For each one, the testing process will be described and the objectives at each stage of testing explained.

Finally, we will look at the different types of testing that can take place throughout the development life cycle.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions at the start of the chapter, the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

Software development lifecycle models (K2)

- FL-2.1.1 Explain the relationships between software development activities and test activities in the software development lifecycle.
- FL-2.1.2 Identify reasons why software development lifecycle models must be adapted to the context of project and product characteristics (K1).

Test levels (K2)

- FL-2.2.1 Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities.

Test types (K2)

- FL-2.3.1 Compare functional, non-functional, and white-box testing.
- FL-2.3.2 Recognize that functional, non-functional, and white-box tests occur at any test level. (K1)
- FL-2.3.3 Compare the purposes of confirmation testing and regression testing.

Maintenance testing (K2)

- FL-2.4.1 Summarize triggers for maintenance testing.
- FL-2.4.2 Describe the role of impact analysis in maintenance testing.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K2)
Which of the following is true of the V model?

a. Coding starts as soon as each function in a system has been defined.
b. The test activities occur after all development activities have been completed.
c. It enables the production of a working version of the system as early as possible.
d. It enables test planning to start as early as possible.

Question SA2 (K2)
Which of the following is true of white-box testing?

a. It is carried out only by developers.
b. It can be used to test data file structures.
c. It is used only at unit and integration test levels.
d. Coverage achieved using white-box test techniques is not measurable.

Question SA3
Which of the following is a test object for integration testing?

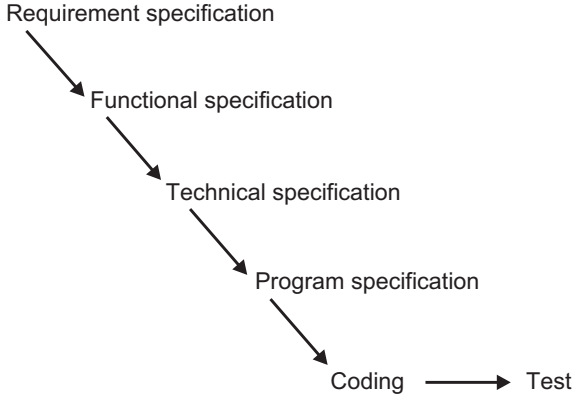
a. A sub-system.
b. An epic.
c. A risk analysis report.
d. A sequence diagram.

SOFTWARE DEVELOPMENT MODELS

A development life cycle for a software product involves capturing the initial requirements from the customer, expanding on these to provide the detail required for code production, writing the code and testing the product, ready for release.

A simple development model is shown in Figure 2.1. This is known traditionally as the waterfall model.

Figure 2.1 Waterfall model



The waterfall model in Figure 2.1 shows the steps in sequence, where the customer requirements are progressively refined to the point where coding can take place. This type of model is often referred to as a linear or sequential model. Each work product or activity is completed before moving on to the next.

In the waterfall model, testing is carried out once the code has been fully developed. Once this is completed, a decision can be made on whether the product can be released into the live environment.

This model for development shows how a fully tested product can be created, but it has a significant drawback: what happens if the product fails the tests? Let us look at a simple case study.

CASE STUDY – DEVELOPMENT PROCESS

Let us consider the manufacture of a smartphone. Smartphones have become an essential part of daily life for many. They must be robust enough to withstand the rigours of being thrown into bags or on floors and must be able to respond quickly to commands.

Many phones now have touchscreens. This means that the apps on the phone must be accessible via a tap on the screen. This is done via a touchscreen driver. The driver is a piece of software that sits between the screen (hardware) and the apps (software), allowing the app to be accessed from a tap on an icon on the screen.

If a waterfall model were to be used to manufacture and ship a touchscreen phone, then all functionality would be tested at the very end, just prior to shipping.

If it is found that the phone can be dropped from a reasonable height without breaking, but that the touchscreen driver is defective, then the phone will have failed in its core required functionality. This is a very late stage in the life cycle to uncover such a fault.

In the waterfall model, the testing at the end serves as a quality check. The product can be accepted or rejected at this point. In the smartphone manufacturing example, this model could be adopted to check that the phone casings after manufacture are crack free, rejecting those that have failed.

In software development, however, it is unlikely that we can simply reject the parts of the system found to be defective and release the rest. The nature of software functionality is such that removal of software is often not a clear-cut activity – this action could cause other areas to function incorrectly. It might even cause the system to become unusable. If the touchscreen driver is not functioning correctly, then some of the apps might not be accessible via a tap on the icon. On a touchscreen phone, this would be an intolerable fault in the live environment.

What is needed is a process that assures quality throughout the development life cycle. At every stage, a check should be made that the work product for that stage meets its objectives. This is a key point: work product evaluation taking place at the point where the product has been declared complete by its creator. If the work product passes its evaluation (test), we can progress to the next stage in confidence. In addition, finding problems at the point of creation should make fixing any problems cheaper than fixing them at a later stage. This is the cost escalation model, described in **Chapter 1**.

The checks throughout the life cycle include verification and validation.

Verification – checks that the work product meets the requirements set out for it. An example of this is to ensure that a website being built follows the guidelines for making websites usable by as many people as possible. Verification helps to ensure that we are building the product in the right way.

Validation – changes the focus of work product evaluation to evaluation against user needs. This means ensuring that the behaviour of the work product matches the customer needs as defined for the project. For example, for the same website above, the guidelines may have been written with people familiar with websites in

mind. It may be that this website is also intended for novice users. Validation would include these users checking that they too can use the website easily. Validation helps to ensure that we are building the right product as far as the users are concerned.

There are two types of development model that facilitate early work product evaluation.

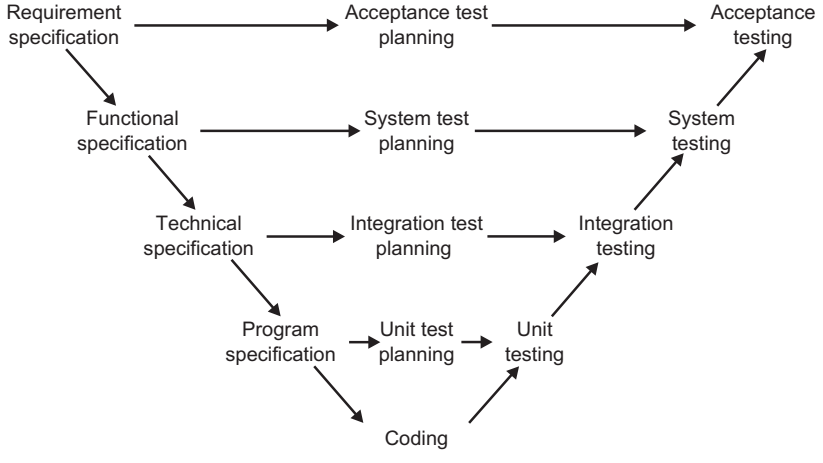
The first is an extension to the waterfall model, known as the V model. The second is a cyclical model, where the coding stage often begins once the initial user needs have been captured. Cyclical models are often referred to as iterative models.

We will consider first the V model.

V model (sequential development model)

There are many variants of the V model. One of these is shown in Figure 2.2.

Figure 2.2 V model for software development



As for the waterfall model, the left-hand side of the model focuses on elaborating the initial requirements, providing successively more technical detail as the development progresses. In the model shown, these are:

- Requirement specification – capturing of user needs.
- Functional specification – definition of functions required to meet user needs.

- Technical specification – technical design of functions identified in the functional specification.
- Program specification – detailed design of each module or unit to be built to meet required functionality.

These specifications could be reviewed to check for the following:

- Conformance to the previous work product (so in the case of the functional specification, verification would include a check against the requirement specification).
- That there is sufficient detail for the subsequent work product to be built correctly (again, for the functional specification, this would include a check that there is sufficient information in order to create the technical specification).
- That it is testable – is the detail provided sufficient for testing the work product?

Formal methods for reviewing documents are discussed in **Chapter 3**.

The middle of the V model shows that planning for testing should start with each work product. Thus, using the requirement specification as an example, acceptance testing is planned for, right at the start of the development. Test planning is discussed in more detail in **Chapter 5**.

The right-hand side focuses on the testing activities. For each work product, a testing activity is identified. These are shown in Figure 2.2:

- Testing against the requirement specification takes place at the acceptance testing stage.
- Testing against the functional specification takes place at the system testing stage.
- Testing against the technical specification takes place at the integration testing stage.
- Testing against the program specification takes place at the unit testing stage.

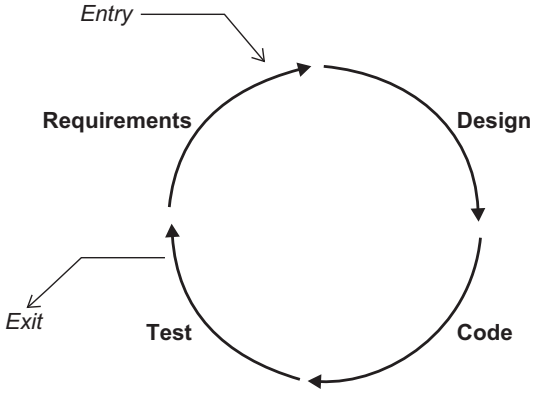
This allows testing to be concentrated on the detail provided in each work product, so that defects can be identified as early as possible in the life cycle, when the work product has been created. The different stages of testing are discussed later.

Remembering that each stage must be completed before the next one can be started; this approach to software development pushes validation of the system by the user representatives right to the end of the life cycle. If the customer needs were not captured accurately in the requirement specification, or if they change, then these issues may not be uncovered until the user testing is carried out. As we saw in **Chapter 1**, fixing problems at this stage could be very costly; in addition, it is possible that the project could be cancelled altogether.

Iterative–incremental development models

Let us now look at a different model for software development – iterative development. This is one where the requirements do not need to be fully defined before coding can start. Instead, a working version of the product is built, in a series of stages, or iterations – hence the name iterative/incremental development. Each stage encompasses requirements definition, design, code and test. This is shown diagrammatically in Figure 2.3.

Figure 2.3 Iterative development



This type of development is often referred to as cyclical – we go 'round the development cycle a number of times', within the project. The project will have a defined timescale and cost. Within this, the cycles will be defined. Each cycle will also have a defined timescale and cost. The cycles are commonly referred to as time-boxes. For each time-box, a requirement is defined and a version of the code is produced, which will allow testing by the user representatives. At the end of each time-box, a decision is made on what extra functionality needs to be created for the next iteration. This process is then repeated until a fully working system has been produced.

Some models incorporate the idea of 'self-organising' teams. This does not mean that the team is leaderless, rather that the team decides how to best manage and execute the tasks amongst themselves. This will of course include the relationship between testers and developers, and how defects are reported.

A key feature of this type of development is the involvement of user representatives in the testing. Having the users represented throughout minimises the risk of developing an unsatisfactory product. The user representatives are empowered to request changes to the software, to meet their needs.

Components or systems developed using these methods often involve overlapping and iterating test levels throughout development. Ideally, each feature is tested at several

test levels before delivery. This is often facilitated by continuous delivery or deployment, enabled by making use of significant automation.

This approach to software development can pose problems, however.

The lack of formal documentation can make it difficult to test. To counter this, developers may use test-driven development (TDD). This is where functional tests are written first, and code is then created and tested. It is reworked until it passes the tests.

In addition, the working environment may be such that developers make any changes required, without formally recording them. This approach could mean that changes cannot be traced back to the requirements, nor to the parts of the software that have changed. Thus, traceability as the project progresses is reduced. To mitigate this, a robust process must be put in place at the start of the project to manage these changes (often part of a configuration management process – this is discussed further in **Chapter 5**).

Another issue associated with changes is the amount of testing required to ensure that implementation of the changes does not cause unintended changes to other parts of the software (this is called regression testing, discussed later in this chapter).

Forms of iterative development include Scrum, Kanban, Spiral and the Rational Unified Process (RUP). Agile is an umbrella term incorporating these and other methods.

- Scrum – here the focus is on short iterations spanning just hours, days or a few weeks. The increments developed are thus correspondingly small. This term may already be familiar to those of you already working in an Agile environment.
- Kanban – as for Scrum, you may already be familiar with this term. It allows for easy visualisation of a workflow, via the usual task board used in Agile development projects. It is not a time-boxing tool; it can be used to show progress of a single enhancement or group of features, from a 'to-do' state, to a 'done' state.
- Rational Unified Process – iterations here tend to be longer than in Scrum, with correspondingly larger feature sets. Those of you working in this environment may recall the Inception – Elaboration – Construction – Transition phases.
- Spiral – Dr Barry Boehm (whom you came across in **Chapter 1** when discussing the cost escalation model) created this model. Here, risk is used as the driver for determining the levels of documentation and effort required for a given project. This can include a prototyping model, where increments created may be reworked significantly or even abandoned if the risks are too high.

Agile methods of developing software have gained significant ground in recent years. Organisations across business sectors have embraced this collaborative way of working and many qualifications focusing on Agile methodologies now exist. The syllabus for this qualification does not dwell on Agile; however, for completeness of learning, a summary will now be provided.

The Agile development methodology is supported through the Agile Alliance, www.agilealliance.org. The Alliance has created an Agile manifesto with four points, supported by 12 principles. The essence of these is to espouse the value of adopting a can-do and collaborative approach to creating a product. The idea is that the development teams work closely with the business, responding to their needs at the time, rather than attempting to adhere to a contract for requirements that might well need to be changed prior to the launch date. Many examples can be provided to suggest that this is a suitable way of working. Going back to our smartphone example, there are many well-known phone manufacturers who failed to move with consumer demands, costing them significant market share.

A popular framework for Agile is Scrum. Scrum is not an acronym; it was taken from the game of rugby. In rugby the team huddles to agree tactics; the ball is then passed back and forth until a sprint to the touchline is attempted. In Scrum, there is a daily stand-up meeting to agree tactics for the day; an agreed set of functions to be delivered at the end of a time-box (Sprint); periodic reviews of functionality by the customer representatives; and a team retrospective to reflect on the previous Sprint in order to improve on the next. In Agile, the term 'user story' is common when referring to requirements, as is the term 'backlog' when referring to a set of requirements or tasks for a particular Sprint.

The ISTQB now offers a qualification in Agile testing as an extension to this Foundation in software testing. Further information can be found at www.istqb.org

CHECK OF UNDERSTANDING

1. What is meant by verification?
2. What is meant by validation?
3. Name three work products typically shown in the V model.
4. Name three activities typically shown in the V model.
5. Identify a benefit of the V model.
6. Identify a drawback of the V model.
7. Name three activities typically associated with an iterative model.
8. Identify a significant benefit of an iterative model.
9. List three challenges of an iterative development.
10. List three types of iterative development.
11. Compare the work products in the V model with those in an iterative model.

For both types of development, testing plays a significant role. Testing helps to ensure that the work products are being developed in the right way (verification) and that the product will meet the user needs (validation).

Characteristics of good testing across the development life cycle include:

- Early test design – in the V model, we saw that test planning begins with the specification documents. This activity is part of the test process, discussed in **Chapter 1**. After test planning, the documents are analysed and test cases designed. This approach ensures that testing starts with the development of the requirements; that is, a proactive approach to testing is undertaken. Proactive approaches to test design are discussed further in **Chapter 5**. As we saw in iterative development, test-driven development may be adopted, pushing testing to the front of the development activity.
- Each work product is tested – in the V model, each document on the left is tested by an activity on the right. Each specification document is called the test basis, that is, it is the basis on which tests are created. In iterative development, the functionality for each iteration is tested before moving on to the next.
- Each test level has objectives specific to that level. Thus, at unit level the focus is on individual pieces of code; at integration level the focus is on the interfaces and so on.
- Testers are involved in reviewing requirements before they are released – in the V model, testers are invited to review associated documents from a testing perspective. Techniques for reviewing documents are outlined in **Chapter 3**.

TEST LEVELS

In Figure 2.2, the test stages of the V model are shown. They are often called test levels. The term test level provides an indication of the focus of the testing, and the types of problems it is likely to uncover. The typical levels of testing are:

- component (unit) testing;
- integration testing;
- system testing;
- acceptance testing.

Each of these test levels will include tests designed to uncover problems specifically at that stage of development. These levels of testing can also be applied to iterative development. In addition, the levels may change depending on the system. For instance, if the system includes some software developed by external parties, or bought off the shelf (commercial off-the shelf (COTS) based), acceptance testing on these may be conducted before testing the system as a whole.

Each test level will have a test basis (a description of the item) and a test object (the item under test). A test basis is some form of definition of what the code is intended to do and is used as a reference for deriving the tests. It can include the requirements; user stories; the source code; or the knowledge of the tester, based on experience. The higher

the level of documentation, the more precise the test design can be. Typically, in V model development, there is more documentation than in iterative development. Techniques for test design will be covered in **Chapter 4**.

Test levels are characterised by the following attributes:

- specific objectives;
- test basis, referenced to derive test cases;
- test object (i.e. what is being tested);
- typical defects and failures;
- specific approaches and responsibilities.

Let us now look at these levels of testing in more detail.

Component (unit) testing

Before testing of the code can start, clearly the code has to be written. This is shown at the bottom of the V model. Generally, the code is written in component parts, or units. The components are usually constructed in isolation, for integration at a later stage. Components are also called programs, modules or units.

Component (unit) testing is often done in isolation from the rest of the system, depending on the Software Development Life Cycle model and the system, which may require mock objects, service virtualisation, harnesses, stubs and drivers.

Component (unit) testing may cover:

- Functional requirements (such as the ability to remove items from a shopping cart).
- Non-functional characteristics (such as checking for memory leaks – this is where the program holds on to memory it is no longer using, which may cause the system to slow down when in use).
- Structural testing – this is checking the percentage of code exercised through testing. Testing based on code (white-box testing) is discussed in **Chapter 4**.

Component (unit) testing is intended to check the quality of the individual piece of code prior to its integration with other units.

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the component are as designed and specified;
  - building confidence in the component's quality;
  - finding defects in the component;
  - preventing defects from escaping to higher test levels.

- Test bases include:
  - a detailed design;
  - a component specification;
  - a data model or other document describing the expected functionality of the unit;
  - the code itself can be used as a basis for component testing.
- Test objects include:
  - the components;
  - the programs;
  - code and data structures;
  - classes;
  - database modules and other pieces of code.
- Typical defects and failures include:
  - incorrect functionality, perhaps due to incorrect logic – for example introducing too short a time for displaying an error message to a user before removing it, causing them not to see it properly;
  - data flow problems – for instance, a part of the code requests an input, but provides no output;
  - code that contains overly complicated constructs, reducing maintainability of the code.
- Specific approaches and responsibilities:
  - Developers tend to fix defects are soon as they are found.
  - One approach to unit testing is called test-driven development. This originated in eXtreme Programming. As its name suggests, test cases are written first, and then the code is built, tested and changed until the unit passes its tests. This is an iterative approach to unit testing.
  - Unit testing is often supported by a unit test framework, as well as debugging tools. These assist the developer in finding and fixing defects, without the need for a formal defect management process at this stage. If defects are logged and analysed however, they can provide opportunities for root cause analysis to improve the test process for future releases.
  - Regression testing is automated so that issues with a software build can be detected quickly. This is now typical in iterative development models.

Integration testing

Once the units have been written, the next stage is to put them together to create the system. This is called integration. It involves building something larger from a number of smaller pieces.

The purpose of integration testing is to expose defects in the interfaces and in the interactions between integrated components or systems.

There are two different levels of integration testing described in the ISTQB syllabus, which may be carried out on test objects of varying size as follows:

Component integration testing, which focuses on the interactions and interfaces between integrated components. It is performed after component testing and is generally automated. In iterative and incremental development, component integration tests are usually part of the continuous integration process.

System integration testing, which focuses on the interactions and interfaces between systems, packages and microservices (where an application is decomposed into fine-grained services, loosely coupled to make up the system). It can also cover interactions and interfaces with external organisations. For example, a trading system in an investment bank may interact with the stock exchange to get the latest prices for its stocks and shares on the international market. Where external organisations are involved, extra challenges for testing present themselves, since the developing organisation will not have control over the interfaces. This can include creating the test environment, defect resolution and so on.

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the interfaces are as designed and specified;
  - building confidence in the quality of the interfaces;
  - finding defects (which may be in the interfaces themselves or within the components or systems);
  - preventing defects from escaping to higher test levels.
- Test bases include:
  - software and system design;
  - sequence diagrams;
  - interface and communication protocol specifications;
  - use cases;
  - architecture at component or system level;
  - workflows;
  - external interface definitions.
- Test objects include:
  - sub-systems;
  - databases;
  - infrastructure;

  - interfaces;
  - Application Program Interfaces (APIs);
  - microservices.
- Typical defects and failures for component integration testing include:
  - incorrect or missing data;
  - incorrect data encoding;
  - incorrect sequencing or timing of interface calls;
  - interface mismatches;
  - failures in communication between components;
  - ignored or improperly handled communication failures between components;
  - incorrect assumptions about the meaning, units or boundaries of the data being passed between components.
- Typical defects and failures for system integration testing include:
  - inconsistent message structures between systems;
  - incorrect or missing data;
  - incorrect data encoding (as above) for component integration testing;
  - interface mismatch (as above) for component integration testing;
  - failures in communication between systems;
  - ignored or improperly handled communication failures between systems;
  - incorrect assumptions about the meaning, units or boundaries of the data being passed between systems;
  - failure to comply with mandatory security regulations.
- Specific approaches and responsibilities.
  - Component integration testing is usually carried out by developers.
  - System integration testing may be done after system testing or in parallel with ongoing system test activities (in both sequential development and iterative and incremental development). It is usually carried out by testers.
  - Continuous integration, where software is integrated on a component-by-component basis (i.e. functional integration), is now commonplace. This allows integration defects to be found as soon as they are introduced.
  - Regression testing is often automated, as we saw for component testing.

Before integration testing can be planned, an integration strategy is required. This involves making decisions on how the system will be put together in a systematic way prior to testing.

Systematic integration strategies may be based on the system architecture (e.g. top-down and bottom-up), functional tasks, transaction processing sequences or some other aspect of the system or components.

There are three commonly quoted integration strategies, as follows.

Big-bang integration

This is where all units are linked at once, resulting in a complete system. When the testing of this system is conducted, it is difficult to isolate any errors found because attention is not paid to verifying the interfaces across individual units.

This type of integration is generally regarded as a poor choice of integration strategy. It introduces the risk that problems may be discovered late in the project, when they are more expensive to fix.

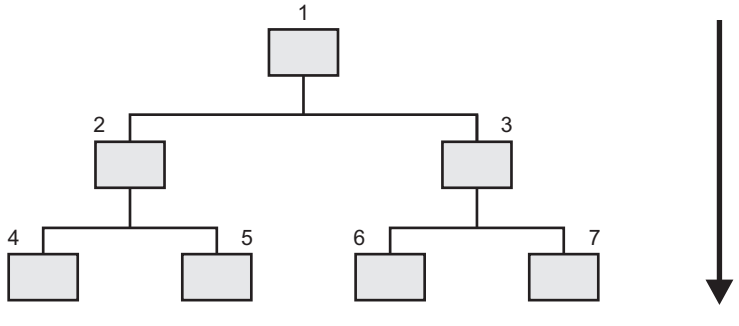
Top-down integration

This is where the system is built in stages, starting with components that, when activated, cause other components to become active. These are called 'calling' components. Components that call others are usually placed above those that are called. Top-down integration testing permits the tester to evaluate component interfaces, starting with those at the 'top'.

Let us look at the diagram in Figure 2.4 to explain this further.

The control structure of a program can be represented in a chart. In Figure 2.4, component 1 can call components 2 and 3. Thus in the structure, component 1 is placed above components 2 and 3. Component 2 can call components 4 and 5. Component 3 can call components 6 and 7. Thus in the structure, components 2 and 3 are placed above components 4 and 5 and components 6 and 7, respectively.

Figure 2.4 Top-down control structure



In this chart, the order of integration might be:

- 1,2
- 1,3
- 2,4
- 2,5
- 3,6
- 3,7

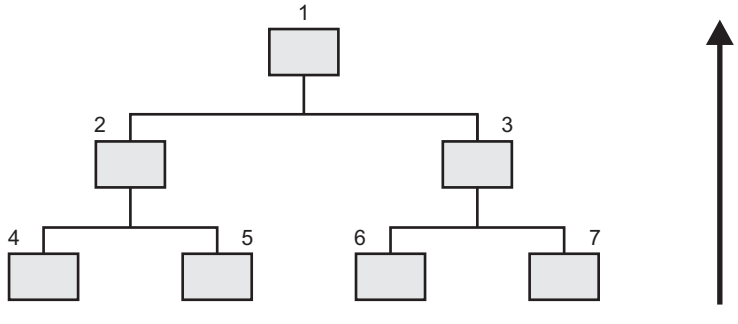
Top-down integration testing requires that the interactions of each component must be tested when they are built. Those lower down in the hierarchy may not have been built or integrated yet. In Figure 2.4, in order to test component 1's interaction with component 2, it may be necessary to replace component 2 with a substitute since component 2 may not have been integrated yet. This is done by creating a skeletal implementation of the component, called a stub. A stub is a passive component, called by other components. In this example, stubs may be used to replace components 4 and 5 when testing component 2.

The use of stubs is commonplace in top-down integration, replacing components not yet integrated.

Bottom-up integration

This is the opposite of top-down integration and the components are integrated in a bottom-up order. This is shown in Figure 2.5.

Figure 2.5 Bottom-up integration



The integration order might be:

- 4,2
- 5,2

- 6,3
- 7,3
- 2,1
- 3,1

So, in bottom-up integration, components 4–7 would be integrated before components 2 and 3. In this case, the components that may not be in place are those that actively call other components. As in top-down integration testing, they must be replaced by specially written components. When these special components call other components, they are called drivers. They are so called because, in the functioning program, they are active, controlling other components. Components 2 and 3 could be replaced by drivers when testing components 4–7. They are generally more complex than stubs.

System testing

Having checked that the components all work together at unit integration level, the next step is to consider the functionality from an end-to-end perspective. This activity is called system testing.

System testing is necessary because many of the criteria for test selection at unit and integration testing result in the production of a set of test cases that are unrepresentative of the operating conditions in the live environment. Thus, testing at these levels is unlikely to reveal errors due to interactions across the whole system, or those due to environmental issues.

System testing serves to correct this imbalance by focusing on the behaviour of the whole system/product as defined by the scope of a development project or programme, in a representative live environment. It is usually carried out by a team that is independent of the development process. The benefit of this independence is that an objective assessment of the system can be made, based on the specifications as written, and not the code.

System testing often produces information that is used by stakeholders to make release decisions, which may include checking that legal or regulatory requirements and standards have been met.

The behaviour required of the system may be documented in functional specifications, use cases or user stories. These should include the functional and non-functional requirements of the system or feature.

A functional requirement is a requirement that specifies a function that a system or system component must perform. Functional requirements can be specific to a system. For instance, you expect to be able to search for flights on a travel agent's website, whereas you visit your online bank to check that you have sufficient funds to pay for the flight. Thus, functional requirements provide detail on what the application being developed will do.

Non-functional system testing looks at those aspects that are important but not directly related to what functions the system performs. These tend to be generic requirements, which can be applied to many different systems. In the example above, you can expect that both systems will respond to your inputs in a reasonable time frame, for instance. Typically, these requirements will consider both normal operations and behaviour under exceptional circumstances. Thus, non-functional requirements detail how the application will perform in use.

Examples of non-functional requirements include:

- installability – installation procedures;
- maintainability – ability to introduce changes to the system;
- performance – expected normal behaviour;
- load handling – behaviour of the system under increasing load;
- stress handling – behaviour at the upper limits of system capability;
- portability – use on different operating platforms;
- recovery – recovery procedures on failure;
- reliability – ability of the software to perform its required functions over time;
- usability – ease with which users can engage with the system.

The amount of testing required at system testing, however, can be influenced by the amount of testing carried out (if any) at the previous stages. In addition, the amount of testing advisable also depends on the amount of verification carried out on the requirements (this is discussed further in **Chapter 3**).

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the system are as designed and specified;
  - validating that the system is complete and will work as expected;
  - building confidence in the quality of the system as a whole;
  - finding defects;
  - preventing defects from escaping to higher test levels or production.
- Test bases include:
  - system and software requirement specifications (functional and non-functional);
  - risk analysis reports;
  - use cases;
  - epics and user stories;

    - models of system behaviour;
    - state diagrams;
    - system and user manuals.
- Test objects include:
    - applications;
    - hardware/software systems;
    - operating systems;
    - system under test (SUT);
    - system configuration and configuration data (i.e. data that can be configured to suit a particular use or need).
- Typical defects and failures include:
    - incorrect calculations;
    - incorrect or unexpected system behaviour;
    - incorrect control and/or data flows within the system.
- Specific approaches and responsibilities:
    - Testers should be involved in the static review of documents to avoid ambiguities in, and lack of understanding of, requirements. These can lead to the false positives and negatives discussed in **Chapter 1**. Reviews will be explored further in **Chapter 3**.
    - System testing should use the most appropriate techniques (to be covered in **Chapter 4**) for the aspect(s) of the system to be tested.
    - The test environment should mimic the target or production environment as far as practicable.
    - System testing is typically carried out by independent testers.
    - As we saw earlier, automation is often used in regression testing to aid in providing confidence in the eventual system functionality.

Acceptance testing

The purpose of acceptance testing is to provide the end users with confidence that the system will function according to their expectations.

Unlike system testing, however, the testing conducted here should be independent of any other testing carried out. Its key purpose is to demonstrate system conformance to, for example, the customer requirements and operational and maintenance processes. For instance, acceptance testing may assess the system's readiness for deployment and use.

Typical forms of acceptance testing include the following:

- User acceptance testing – testing by user representatives to check that the system meets their business needs. This can include factory acceptance testing, where the system is tested by the users before moving it to their own site. Site acceptance testing could then be performed by the users at their own site.
- Operational acceptance testing – often called operational readiness testing. This involves checking that the processes and procedures are in place to allow the system to be used and maintained. This can include checking:
  - back-up facilities;
  - installing, uninstalling and upgrading;
  - performance testing;
  - procedures for disaster recovery;
  - user management;
  - maintenance procedures;
  - data load and migration tasks;
  - security vulnerabilities.
- Contract and regulatory acceptance testing:
  - Contractual acceptance testing – sometimes the criteria for accepting a system are documented in a contract. Testing is then conducted to check that these criteria have been met, before the system is accepted. This is typical of custom-developed software.
  - Regulatory acceptance testing – in some industries, systems must meet governmental, legal or safety standards. Examples of these are the defence, banking and pharmaceutical industries. The results of tests here may be witnessed or audited by regulatory bodies.
- Alpha and beta testing:
  - Alpha testing takes place at the developer's site – the operational system is tested while still at the developer's site by internal staff, before release to external customers. Note that testing here is still independent of the development team.
  - Beta testing takes place at the customer's site – the operational system is tested by a group of customers, who use the product at their own locations and provide feedback, before the system is released. This is often called 'field testing'.

Both alpha and beta testing are typically used by developers of COTS software in order to get feedback before final go-live.

- Specific objectives:
  - establishing confidence in the quality of the system as a whole;
  - validating that the system is complete and will work as expected;
  - verifying that functional and non-functional behaviours of the system are as specified.

It is worth noting that defect finding is not a main aim of acceptance testing, although they must of course be logged and resolved when found.

- Test bases include:
  - user, business, legal, regulatory and system requirements;
  - use cases and business processes;
  - installation procedures;
  - risk analysis reports.
- Test bases for operational acceptance testing include:
  - back-up, restore and disaster recovery procedures;
  - security standards or regulations;
  - non-functional requirements;
  - operations documentation;
  - deployment and installation instructions;
  - performance targets;
  - database packages.
- Test objects include:
  - system under test;
  - system configuration, configuration and production data;
  - business processes for a fully integrated system;
  - recovery systems and hot sites (for business continuity and disaster recovery testing);
  - operational and maintenance processes;
  - forms and reports.
- Typical defects and failures.
  - System workflows do not meet business or user requirements.
  - Business rules are not implemented correctly.
  - System does not satisfy contractual or regulatory requirements.
  - Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform.
- Specific approaches and responsibilities.
  - Acceptance testing is often the responsibility of the customers or users of a system, although other project team members may be involved as well.
  - Acceptance testing is often thought of as the last test level in a sequential development life cycle, but it may also occur at other times; for example:

- When a COTS software product is installed or integrated.
- Before system testing for a new functional enhancement.
- At the end of each iteration in iterative development – for verification against the documented acceptance criteria and validation against the user needs. This can also include alpha and beta testing.

In iterative development, project teams can employ various forms of acceptance testing during and at the end of each iteration, such as those focused on verifying a new feature against its acceptance criteria and those focused on validating that a new feature satisfies the users' needs. In addition, alpha tests and beta tests may occur, either at the end of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests and contractual acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

CHECK OF UNDERSTANDING

1. List two documents that could be used as the test basis for unit testing.
2. Describe test driven development (TDD).
3. Identify two typical test objects used for integration testing.
4. List three documents used as the test basis for system testing.
5. Compare a functional requirement with a non-functional requirement.
6. What is the purpose of acceptance testing?
7. List three documents used as a test basis for acceptance testing.
8. Identify three types of acceptance testing.

TEST TYPES

In the previous section we saw that each test level has specific testing objectives. In this section we will look at the types of testing required to meet these objectives.

Test types fall into the following categories:

- functional testing;
- non-functional testing;
- white-box testing;
- testing after code has been changed.

Functional testing

As you saw in the section on system testing, functional testing looks at the specific functionality of a system, such as searching for flights on a website, or perhaps calculating employee pay correctly using a payroll system. This can include checks for completeness, correctness and appropriateness.

Functional testing is carried out at all levels of testing, from unit through to acceptance testing. In the example above, the testing of the employee pay may be done during unit testing; whereas searching for a flight is often done during system testing.

Functional testing is also called specification-based testing or black-box testing (covered in **Chapter 4**). It can be measured in terms of the percentage of requirements covered by the tests.

Designing tests at this level often requires specific domain skills. In today's world, testing the blockchain used in crypto-currencies requires different knowledge and skills to those used in designing tests for normal banking operations, for instance.

Non-functional testing

This is where the behavioural aspects of the system are tested. As you saw in the section on system testing, examples include usability, performance, efficiency and security testing among others.

As for functional testing, non-functional testing:

- should be performed at all levels so that potential defects are detected as early as possible.
- can make use of black-box testing techniques, such as checking that a flight can be booked within a specific time frame;
- often requires specialist knowledge (such as knowing the inherent weaknesses of specific technologies) and skills (such as having an understanding of how to carry out performance testing);
- can be measured – for instance checking the percentage of mobile devices tested for compatibility with an application.

These tests can be referenced against a quality model, such as the one defined in ISO/IEC 25010 Systems and software Quality Requirements and Evaluation (SQuaRE). Note that a detailed understanding of this standard is not required for the exam.

White-box testing

In white-box testing our focus is on the internal structure of the system. This could be the code itself, an architectural definition or data flows through the system.

White-box testing is commonly carried out at unit and component integration test levels. Here, common measures include code and interface coverage (percentage of code and

interfaces exercised by tests). Further detail on code coverage measures is provided in **Chapter 4**.

It can also be carried out at the higher levels of testing where a structural definition of the system exists. An example is a business flow (represented as a flow chart), which could be used to design tests at system or higher levels.

As before, this type of testing also requires specialised knowledge and skills, such as code creation, data storage on databases and use of the associated tools.

Testing related to changes

The previous sections detail the testing to be carried out at the different stages in the development life cycle. At any level of testing, it can be expected that defects will be discovered. When these are found and fixed, the quality of the system being delivered is improved.

After a defect is detected and fixed, the changed software should be retested to confirm that the problem has been successfully removed. This is called retesting or confirmation testing. Note that when the developer removes the defect this activity is called debugging, which is not a testing activity. Testing finds a defect, debugging fixes it.

The unchanged software should also be retested to ensure that no additional defects have been introduced as a result of changes to the software. This is called regression testing. Regression testing should also be carried out if the environment has changed.

Regression testing involves the creation of a set of tests, which serve to demonstrate that the system works as expected. These are run many times over a testing project, when changes are made, as discussed above. This repetition of tests makes regression testing suitable for automation in many cases. Test automation is covered in detail in **Chapter 6**.

In iterative development projects such as Agile development, the requirements churn introduces a great need for both confirmation and regression testing. There is also a concept of code refactoring (where a developer seeks to increase the quality of the code written), which also necessitates change-related testing.

CHECK OF UNDERSTANDING

Which of the following is correct?

a. Regression testing checks that a problem has been successfully addressed, while confirmation testing is done at the end of each release.
b. Regression testing checks that all problems have been successfully addressed, while confirmation testing refers to testing individual fixes.
c. Regression testing checks that fixes to errors do not introduce unexpected functionality into the system, while confirmation testing checks that fixes have been successful.
d. Regression testing checks that all required testing has been carried out, while confirmation testing checks that each test is complete.

MAINTENANCE TESTING

For many projects (though not all) the system is eventually released into the live environment. Hopefully, once deployed, it will be in service as long as intended, perhaps for years or decades.

During this deployment, it may become necessary to change the system.

Triggers for maintenance include:

- additional features being required;
- the system being migrated to a new operating platform;
- the system being retired – data may need to be migrated or archived;
- planned upgrade to COTS-based systems;
- new faults being found requiring fixing (these can be 'hot fixes').

Once changes have been made to the system, they will need to be tested (retesting), and it also will be necessary to conduct regression testing to ensure that the rest of the system has not been adversely affected by the changes. Testing that takes place on a system that is in operation in the live environment is called maintenance testing.

When changes are made to migrate from one platform to another, the system should also be tested in its new environment. When migration includes data being transferred in from another application, then conversion testing also becomes necessary.

As we have suggested, all changes must be tested, and, ideally, all of the system should be subject to regression testing. In practice, this may not be feasible or cost-effective. An understanding of the parts of the system that could be affected by the changes could

reduce the amount of regression testing required. Working this out is termed impact analysis; that is, analysing the impact of the changes on the system.

Impact analysis for maintenance

The purpose of impact analysis is to determine the likely impact of a change to a system. We need to understand the intentions of the change, any potential side effects of the change, and how existing tests may need to be changed.

This can be difficult for a system that has already been released and is in maintenance. This is because the specifications may be out of date (or non-existent); test cases may have not been documented; there is a lack of traceability of tests backs to requirements; there is weak or non-existent tool support; or the original development team may have moved on to other projects or left the organisation altogether.

CHECK OF UNDERSTANDING

1. How do functional requirements differ from non-functional requirements?
2. For which type of testing is code coverage measured?
3. What is the purpose of maintenance testing?
4. Give examples of when maintenance testing is necessary.
5. What is meant by the term impact analysis?

SUMMARY

In this chapter we have explored the role of testing within the Software Development Life Cycle. We have looked at the basic steps in any development model, from understanding customer needs to delivery of the final product. These were built up into formally recognisable models, using distinct approaches to software development.

The V model, as we have seen, is a stepwise approach to software development, meaning that each stage in the model must be completed before the next stage can be started, if a strict implementation of the model is required. This is often the case in safety-critical developments. The V model typically has the following work products and activities:

1. requirement specification;
2. functional specification;
3. technical specification;
4. program specification;
5. code;
6. unit testing;
7. integration testing;

8. system testing;
9. acceptance testing.

Work products 1–5 are subject to verification, to ensure that they have been created following the rules set out. For example, the program specification is assessed to ensure that it meets the requirements set out in the technical specification, and that it contains sufficient detail for the code to be produced.

In activities 6–9, the code is assessed progressively for compliance to user needs, as captured in the specifications for each level.

An iterative model for development has fewer steps but involves the user from the start. These steps are typically:

1. define iteration requirement;
2. build iteration;
3. test iteration.

This sequence is repeated for each iteration until an acceptable product has been developed.

An explanation of each of the test levels in the V model was given. For unit testing the focus is the code within the unit itself, for integration testing it is the interfacing between units, for system testing it is the end-to-end functionality, and for acceptance testing it is the user perspective.

An explanation of test types was then given and by combining test types with test levels we can construct a test approach that matches a given system and a given set of test objectives very closely. The techniques associated with test types are covered in detail in **Chapter 4** and the creation of a test approach is covered in **Chapter 5**.

Finally, we looked at the testing required when a system has been released, but a change has become necessary – maintenance testing. We discussed the need for impact analysis in deciding how much regression testing to do after the changes have been implemented. This can pose an added challenge if the requirements associated with the system are missing or have been poorly defined.

In the next chapter, techniques for improving requirements will be discussed.

Example examination questions with answers

E1. K1 question
Which of the following are test levels where white-box testing is applicable?

i. Unit testing.
ii. Acceptance testing.
iii. Regression testing.
iv. Performance testing.

a. i and ii.
b. i only.
c. ii and iii.
d. ii and iv.

E2. K2 question
Which of the following is true of non-functional testing?

a. Examples of non-functional testing are provided in ISO Standard 20246.
b. It is best carried out at system and acceptance test levels.
c. It cannot usually be measured.
d. It can make use of black-box test techniques.

E3. K2 question
Which of the following iterative development models tends to work with shorter iterations relative to the others?

a. Waterfall model.
b. Rational Unified Process.
c. Scrum.
d. V model.

E4. K2 question
Which of the following statements are true?

i. For every development activity there is a corresponding testing activity.
ii. Each test level has the same test objectives.
iii. The analysis and design of tests for a given test level should begin after the corresponding development activity.
iv. Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.

a. i and ii.
b. iii and iv.
c. ii and iii.
d. i and iv.

E5. K2 question
Which of the following is not true of regression testing?

a. It can be carried out at each stage of the life cycle.
b. It serves to demonstrate that the changed software works as intended.
c. It serves to demonstrate that software has not been unintentionally changed.
d. It is often automated.

Answers to questions in the chapter

SA1. The correct answer is d.
SA2. The correct answer is b.
SA3. The correct answer is a.

Answers to example examination questions

E1. The correct answer is a.
White-box testing is applicable at all test levels. Regression and performance testing are not test levels; they are test types.

E2. The correct answer is d.
Option a provides a standard for use in reviews. The standard used for non-functional testing is ISO 25010. Option b is incorrect – non-functional testing should be carried out all levels. Option c is incorrect, it can be measured in terms of percentage of non-functional requirements covered.

E3. The correct answer is c.
Options a and d are sequential models and do not use iterative development. Option b – the Rational Unified Process – tends to use longer iterations than Scrum.

E4. The correct answer is d.
Option ii is incorrect – each test level has a different objective. Option iii is also incorrect– test analysis and design should start once the documentation has been completed.

E5. The correct answer is b.
This is a definition of confirmation testing. The other three options are true of regression testing.

3 STATIC TESTING

Geoff Thompson

INTRODUCTION

This chapter provides an introduction to an important area of software testing – static testing. Static testing techniques test software without executing it. They are important because they can find errors and defects before code is built/executed and therefore earlier in the life cycle of a project, making corrections easier and cheaper to achieve than for the same defects found during test execution. Review types and techniques are central to the static testing approach, and in this chapter we will look at the alternative types of review and how they fit with the test process that was defined in **Chapter 1**.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions that follow the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

Static testing basics (K2)

- FL-3.1.1 Recognize types of software work product that can be examined by the different static testing techniques. (K1)
- FL-3.1.2 Use examples to describe the value of static testing.
- FL-3.1.3 Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software lifecycle.

Review process

- FL-3.2.1 Summarize the activities of the work product review process.
- FL-3.2.2 Recognize the different roles and responsibilities in a formal review. (K1)

- FL-3.2.3 Explain the differences between different review types: informal review, walkthrough technical review, and inspection. (K2)
- FL-3.2.4 Apply a review technique to a work product to find defects. (K3)
- FL-3.2.5 Explain the factors that contribute to a successful review. (K2)

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K1)
One of the roles in a review is that of facilitator. Which of the following best describes this role?

a. Ensures the effective running of the review meetings, where it is decided that a meeting is required.
b. Allocates time in the plan, decides which reviews will take place and ensures that the benefits are delivered.
c. Writes the document to be reviewed, agrees that the document can be reviewed and updates the document with any changes.
d. Documents all issues raised in the review meeting, records problems and open points.

Question SA2 (K2)
Which of the following statements are correct for walkthroughs?

i. Often led by the author.
ii. Documented and defined results.
iii. All participants have defined roles.
iv. Used to aid learning.
v. Main purpose is to find defects.

a. i, ii and v are correct.
b. ii, iii and iv are correct.
c. i, iv and v are correct.
d. iii, iv and v are correct.

Question SA3 (K1)
Which of the following is an activity in the review process?

a. Design of the document.
b. Booking meeting rooms.
c. Writing program code.
d. Fixing and reporting.

BACKGROUND TO STATIC TESTING

Static testing tests software and work products without executing the code. Typically, this includes requirements or specification documents, and the testing of code without actually executing it. The first of these is known as a review and is typically used to find and remove errors and ambiguities in documents before they are used in the development process, thus reducing one source of defects in the code; the second is known as static analysis, and it enables code to be analysed for structural defects or systematic programming weaknesses that may lead to defects.

Static techniques find the causes of failures rather than the failure itself, which is found during test execution.

Reviews are normally completed manually; static analysis is normally completed automatically using tools.

Giving a draft document to a colleague to read is the simplest example of a review, and one that can yield a larger crop of errors than often anticipated (see **Chapter 5** regarding 'world view' to understand why).

WORK PRODUCTS THAT CAN BE EXAMINED BY STATIC TESTING

Virtually any work product can be examined using static testing techniques. A work product is anything that is written down and can include:

- specifications such as business requirements, functional requirements, security requirements and non-functional requirements;
- epics, user stories and acceptance criteria;
- architecture and design specifications;
- code;
- testware such as test plans and test cases;
- user guides;
- web pages;
- contracts, plans, schedules and budgets;
- design models.

BENEFITS OF STATIC TESTING

The earlier in the life cycle that static testing is applied, the larger the benefits are. If work products are statically tested before any code is written, this will remove defects from the work product and ensure these defects are not built into the code. As has been shown in **Chapter 1** the earlier a defect is found, the cheaper it is to fix. If a defect in the design documentation is built into code and even deployed to the user base, the fix can be enormously expensive. How much more expensive a defect found in live use

could be, can be seen in the following example: a bank develops a letter to inform their clients of a change in interest rates. A small defect (spelling mistake), if found in the initial letter design before the letter is coded, costs no more than a few seconds of the author's time to correct. However, if that spelling mistake moves from the design, into the coded letter and goes into live use without being spotted, the cost to fix it increases considerably. The letter needs to be taken out of live use, corrected and retested, which could take up to a week to complete but could have been fixed in a few seconds if found earlier.

Static testing benefits may include:

- detecting and correcting defects more efficiently, and prior to test execution;
- identification of defects not easily found by dynamic testing;
- preventing defects in design or coding by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies and redundancies in requirements;
- increasing development productivity;
- reducing development cost and time;
- reducing dynamic testing cost and time;
- reducing total cost of quality over the software's lifetime, due to fewer failures later in the life cycle or after delivery into live operation;
- improving communication between team members in the course of participating in reviews.

Benefits recognition is key to ensuring that static testing remains a focus, as when no one recognises any benefits there is a better than average chance that static testing will be removed from the project plans. It is important, therefore, that results of static testing are made clear to all stakeholders regularly.

DIFFERENCES BETWEEN STATIC AND DYNAMIC TESTING

Static and dynamic testing have the same objective: to find defects as soon as possible. The main difference is that static testing is carried out against work products without actually executing any code, whereas dynamic testing is carried out by executing actual code or the final software or hardware product.

Often, code may not be exercised that often, or is deeply embedded, so building a dynamic test to find defects is sometimes too hard or impossible to do. Static testing is most effective when it is used to find defects before any code is written, thereby ensuring that the code that is written is not based on wrong or faulty specifications and so on. Static testing can often be used in this situation to find defects quicker and more easily than in dynamic testing.

Static testing improves the consistency and quality of work products, while dynamic testing focuses on externally visible behaviours; perhaps the next screen that is displayed, or the customer details displayed on the screen.

Typical defects, that are easier and cheaper to find during static testing, include:

- requirements defects (e.g. inconsistencies, ambiguities, contradictions, omissions, inaccuracies and redundancies);
- design defects (inefficient algorithms or database structures, high coupling (the lack of interdependence between software modules) and low cohesion (associated with undesirable traits such as being difficult to maintain, test, reuse or even understand));
- coding defects (e.g. variables with undefined values, variables that are declared but never used, unreachable code and duplicate code);
- deviations from standards (e.g. lack of adherence to coding standards);
- incorrect interface specifications (e.g. different units of measurement used by the calling system than by the called system);
- security vulnerabilities (e.g. susceptibility to buffer overflows);
- gaps or inaccuracies in test basis traceability or coverage (e.g. missing tests for an acceptance criterion).

Maintainability defects, such as poor reuse of components, code that is difficult to analyse and modify without introducing new defects, improper modularisation or no documentation at all, can all be found using static testing techniques.

REVIEW PROCESS

There are two types of review, informal and formal.

An informal review is identified by the principle that it is not following a defined process and has no formal documented output.

A formal review is identified by team participation, documented results and documented procedures are followed. Often there will also be defined roles.

The decision on the appropriate level of formality for a review is usually based on combinations of the following factors:

- The type of Software Development Life Cycle (there are different approaches to reviews in an Agile project compared to a waterfall project.
- The maturity of the development process. The more mature the process is, the more formal reviews tend to be.
- The complexity of the work product.
- Legal or regulatory requirements. These are used to govern the software development activities in certain industries, notably in safety-critical areas such as railway signalling, and determine what kinds of review should take place.

- The need for an audit trail. Formal review processes ensure that it is possible to trace backwards throughout the Software Development Life Cycle. The level of formality in the types of review used can help to increase how much is contained within the audit trail.

Reviews can also have a variety of objectives, where the term 'review objective' identifies the main focus for a review. Typical review objectives include:

- finding defects;
- gaining understanding;
- generating discussion;
- education;
- decision making by consensus.

The way a review is conducted will depend on what its specific objective is, so a review aimed primarily at finding defects will be quite different from one that is aimed at gaining understanding of a document.

All reviews, formal and informal alike, exhibit the same basic elements of process:

- The document under review is studied by the reviewers.
- Reviewers identify issues or problems and inform the author either verbally or in a documented form, which might be as formal as raising a defect report or as informal as annotating the document under review.
- The author decides on any action to take in response to the comments, and updates the document accordingly.

This basic process is always present, but in the more formal reviews it is elaborated to include additional stages and more attention to documentation and measurement.

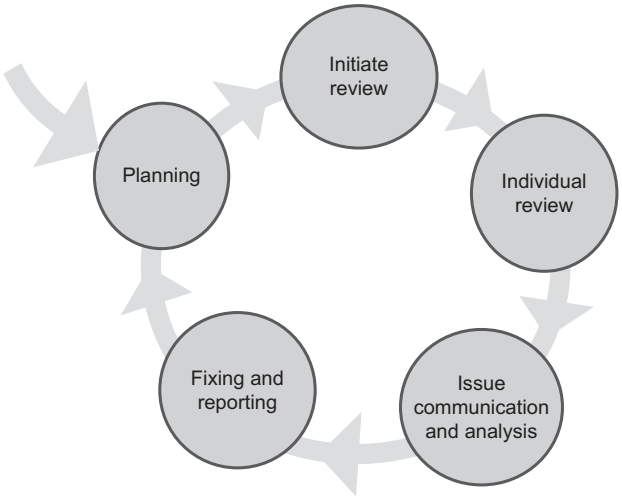
WORK PRODUCT REVIEW PROCESS

Reviews all follow the same basic process: the more formal the review, the more formal the process surrounding the review is. The following list explains the key stages in more detail. See also Figure 3.1 the work product review process flow.

Planning

- Defining the scope – this includes the purpose of the review; for example finding defects, what documents or parts of documents are to be reviewed and the quality characteristics.
- Estimating the effort required and the time frame that the review will be undertaken in – much like the estimating required to fund the test activity, reviews need to be estimated so they can be costed in a project budget.

Figure 3.1 Stages of a formal review



- Identifying the review characteristics such as review type, roles, activities and checklists – as described above, there are many factors that help the decision; for example, what type of review characteristics will be used.
- Selecting the people to undertake the review – ensuring that those selected can and will add value to the process. There is little point in selecting a reviewer who will agree with everything written by the author without question. As a rule of thumb, it is best to include some reviewers who are from a different part of the organisation, who are known to be 'picky' and known to be dissenters. Reviews, like weddings, are enhanced by including 'something old, something new, something borrowed, something blue'. In this case 'something old' would be an experienced practitioner; 'something new' would be a new or inexperienced team member; 'something borrowed' would be someone from a different team; and 'something blue' would be the dissenter who is hard to please. At the earliest stage of the process, a moderator (review leader) must be identified. This is the person who will coordinate all of the review activity. This also includes allocating roles.
- Allocating roles – each reviewer is given a role to provide them with a unique focus on the document under review. Someone in a tester role might be checking for testability and clarity of definition, while someone in a user role might look for simplicity and a clear link to business values. This approach ensures that, although all reviewers are working on the same document, each individual is looking at it from a different perspective.
- Defining the entry and exit criteria, especially for the most formal review types (e.g. inspections) – before a review can start certain criteria have to be met. These are defined by the moderator, and could include: all reviewers must have received the review papers; all reviewers have a kick-off meeting booked in their diaries; any training of reviewers has been completed.

- Checking entry criteria (mainly used for more formal review types such as inspections) – this stage is where the entry criteria agreed earlier are checked to ensure that they have been met so that the review can continue.

Initiate review

- Distributing the work products – the facilitator distributes all of the required documents to all reviewers.
- Explaining the scope, objectives, process, roles and work products to the participants – this can be run as a meeting or simply by sending out the details to the reviewers. The method used will depend on timescales and the volume of information to pass on. A lot of information can be disseminated better by a meeting rather than expecting reviewers to read pages of text.
- Answering any questions raised by the review team.

Individual review

- Reviewing all parts of the work product including the source documents.
- Noting potential defects, questions and comments – in this stage the potential defects, questions and comments found during individual preparation are logged.

Issue communication and analysis

- Communicating potential defects either in a review meeting or via a defect log.
- Analysing potential defects; if agreed that there is a defect, assigning ownership of the repair work.
- Evaluating and documenting the required quality criteria, identifying whether they have been met or not.
- Evaluating the review results against the review exit criteria to decide whether the work product is to be rejected for major change or simply updated with minor changes.

Fixing and reporting

- Creating defect reports for those findings that require changes, which may include making recommendations regarding handling the defects, making decisions about the defects and so on.
- Fixing defects found – here, typically, the author will be fixing defects that were found and agreed as requiring a fix.
- Communicating defects to the appropriate person or team.
- Recording updated status of defects (in formal reviews) – always done during more formal review techniques; optional for others.

- Gathering metrics, such as how much time was spent on the review and how many defects were found, to show notionally how the quality of the reviewed item has increased, but also the value added by the review to later stages of the life cycle; for example, how much the defect would have cost if it hadn't been found until user acceptance testing.
- Checking on exit criteria – the moderator will also check the exit criteria (for more formal review types such as inspections) to ensure that they have been met so that the review can be officially closed.
- Accepting the work product when the exit criteria has been met.

ROLES AND RESPONSIBILITIES

The role of each reviewer is to look at the documents under review (and any appropriate source documents) from their assigned perspective; this may include the use of checklists. For example, a checklist based on a particular perspective (such as user, maintainer, tester or operations) may be used, or a more general checklist (such as typical requirements problems) may be used to identify defects.

In addition to these assigned review roles, the review process itself defines specific roles and responsibilities that should be fulfilled for formal reviews. They are:

- Author: the author is the writer or person with chief responsibility for the development of the document(s) to be reviewed. The author will in most instances also take responsibility for fixing any agreed defects.
- Manager: the manager decides on what is to be reviewed (if not already defined), assigns staff, ensures that there is sufficient time allocated in the project plan for all of the required review activities, monitors ongoing cost-effectiveness of the review, determines if the review objectives have been met, and executes control decisions if objectives are not met. Managers do not normally get involved in the actual review process unless they can add real value; for example, they have technical knowledge key to the review.
- Facilitator (often called a moderator): ensures effective running of the review. The moderator may mediate between the various points of view and is often the person on whom the success of the review rests. The facilitator will also make the final decision as to whether to release an updated document.
- Review leader: the review leader is the person who leads the review of the document or set of documents, including planning the review, running the meeting, and follow-ups after the meeting.
- Reviewers: these are individuals with a specific technical or business background (also called subject matter experts, checkers or inspectors) who, after the necessary preparation, identify and describe findings (e.g. defects) in the product under review. As discussed above, reviewers should be chosen to represent different perspectives and roles in the review process and take part in any review meetings.

- Scribe (or recorder): the scribe attends the review meeting and documents all of the issues and defects, problems and open points that were identified during the meeting.

An additional role not normally associated with reviews is that of the tester. Testers have a particular role to play in relation to document reviews. In their test analysis role, they will be required to analyse a document to enable the development of tests. In analysing the document, they will also review it; for example, in starting to build end-to-end scenarios they will notice if there is a 'hole' in the requirements that will stop the business functioning, such as a process that is missing or some data that is not available at a given point. So, effectively a tester can either be formally invited to review a document or may do so by default in carrying out the tester's normal test analysis role.

CHECK OF UNDERSTANDING

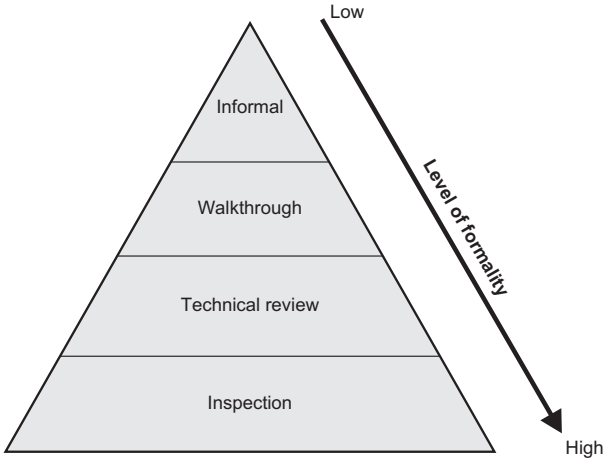
1. Identify three benefits of reviews.
2. What happens during the planning phase of a review?
3. Who manages the review process?

TYPES OF REVIEW

A single document or related work product may be subject to many different review types: for example, an informal review may be carried out before the document is subjected to a technical review or, depending on the level of risk, a technical review or inspection may take place before a walkthrough with a customer.

Figure 3.2 shows the different levels of formality by review type.

Figure 3.2 Formality of reviews



Each type of review has its own defining characteristics. We identify four review types to cover the spectrum of formality. These are usually known as:

1. Informal review (least formal) (e.g. buddy check, pairing, pair review). Key characteristics:

    - The main purpose is detecting potential defects.
    - A possible additional purpose could be to generate new ideas, or to quickly resolve a problem.
    - There is no formal process underpinning the review.
    - It may not involve a review meeting.
    - It may be performed by a colleague of the author or opened up to many people.
    - The review may be documented but this is not required; many informal reviews are not documented.
    - There may be some variations in the usefulness of the review depending on the reviewer; for example, the reviewer does not have the technical skills but is just available to check quickly and ensure that the document makes sense.
    - Use of a checklist is optional.
    - This is a common review in Agile development.

2. Walkthrough. Key characteristics:

    - The main purpose is to find defects, improve the software product, consider alternative implementations, and/or evaluate conformance to standards and specifications.
    - Possible additional purposes include exchanging ideas about techniques or style variations, the training of participants and achieving consensus.
    - Preparation by reviewers before the walkthrough meeting, production of a review report or a list of findings, and appointment of a scribe who is not the author are all optional components.
    - The meeting is led by the author of the document under review and attended by members of the author's peer group.
    - A scribe is mandatory.
    - Use of a checklist is optional.
    - Review sessions are open-ended and may vary in practice from quite informal to very formal.
    - Walkthroughs typically explore scenarios, or conduct dry runs of code or processes.
    - Potential defect logs and review reports may be produced.
    - It may vary in practice from quite informal to very formal.

3. Technical review. Key characteristics:

    - The main purpose is gaining consensus and detecting potential defects.
    - Possible additional purposes are evaluating quality and building confidence in the work product, generating new ideas, motivating authors to improve future work products and to consider alternative implementations.
    - Technical reviews are documented and use a well-defined defect detection process that includes peers and technical experts.
    - Individual preparation before the review meeting is required.
    - A review meeting is optional; if one takes place it is ideally led by a trained facilitator.
    - A scribe is mandatory, ideally not the author.
    - Reviewers using checklists is optional.
    - Potential defect logs and review reports are typically produced.

4. Inspection (most formal). Key characteristics:

    - The main purpose is to detect potential defects, evaluate quality and build confidence in the work product, preventing future similar defects through author learning and root cause analysis.
    - A possible further purpose includes authors improving future work products and the software development process, achieving consensus.
    - The inspection process is formal, based on rules and checklists, and uses entry and exit criteria.
    - Pre-meeting preparation is essential, which includes the reading of any source documents to ensure consistency.
    - Reviewers are either peers of the author or experts in other disciplines that are relevant to the work product.
    - Specific entry and exit criteria are used.
    - A scribe is mandatory.
    - Review meetings are led by a trained facilitator who is not the author and usually involve peer examination of a document; individual inspectors work within defined roles.
    - The author cannot lead the review, or scribe.
    - All potential defects are logged and a review report is produced.
    - Metrics are collected and used to improve the entire software development process, including the inspection process.

In reality, the lines between the review types often get blurred and what is seen as a technical review in one company may be seen as an inspection in another. The above is the 'classic view' of reviews. The key for each company is to agree the objectives and benefits of the reviews that they plan to carry out.

A single work product during its development may be subject to many different types of review.

APPLYING REVIEW TECHNIQUES

As with testing, there are specific techniques that can be used with all of the aforementioned types of review. The effectiveness of the techniques will vary dependant on what type of review they are used on. Examples of review techniques are as follows.

Ad hoc

Used mainly in the less formal review types, in an ad hoc review those involved are provided with little or no guidance on what is expected of them and how the review should be performed. This technique is very dependent on the reviewer's skills and leads to issues like duplication of potential defects identified.

Checklist based

Uses a checklist to deliver a systematic approach to a review, ensuring that the reviewers detect potential defects based on the checklist rules. A review checklist documents the activities to be undertaken, or the types of defects to be identified; for example typos, or differences to source documents or technical content.

Scenarios and dry runs

A reviewer of a scenario and dry runs is given structured guidelines on how to read the document under review. The scenario allows dry runs of products to take place based on expected usage of the work product. A scenario provides better guidelines on how to identify specific defect types and is sometimes seen as better than the checklist approach.

Role based

In a role-based review, the work product is examined from a specific role perspective. Typical roles include end user administrator, system administrator and so on.

Perspective based

A perspective-based review is a little like a role-based review in that the reviewer will take on a different stakeholder's viewpoint. Perspectives such as end user, marketing, designer, tester or operations may be required. This approach leads to more depth in the review as well as a reduction in the duplication of potential defects found.

In addition, this type of review requires the reviewer to attempt to deliver the result of the work product. For example, a tester may have to produce draft acceptance tests if reviewing a requirement specification.

This approach has been shown to be the most effective general review technique.

Success factors for reviews

A successful review will depend on using the right type of review and techniques. There are other factors that can influence the successful outcome of a review. These fall into two types of factors:

1. organisational;
2. people related.

Examples of organisational success factors are:

- Each review should have a clearly predefined and agreed objective and the right people should be involved to ensure that the measurable outcome is met. For example, in an inspection, if the objective is technically based, each reviewer will have a defined role and have the experience to complete the technical review; this should include testers as valued reviewers. Any defects found are welcomed and expressed objectively.
- Review techniques (both formal and informal) that are suitable to the type and level of software work products and reviewers (this is especially important in inspections).
- Review techniques like checklist-based or role-based reviewing provide effective defect identification of the work product being reviewed.
- Checklists are used to ensure focus on areas of main risk and are up to date.
- Reviewing large documents in small chunks, providing frequent feedback on defects as early as possible to the author.
- Review participants have sufficient time to prepare, which may include reading all supporting work products.
- Reviews are scheduled with adequate notice.
- Management support is essential for a good review process (e.g. by incorporating adequate time for review activities in project schedules).

Examples of people-related success factors are:

- The right people are involved to ensure that the objectives are met; for example, people with different skill sets from the relevant user community.
- The use of testers as valued reviewers so that they can learn about the work product, enabling the development of better and more effective tests, and to be able to develop those tests early.
- Adequate time is allocated to each participant.
- Chunking of the work product into smaller sections enables the reviewer to focus and not lose attention during the actual review meeting.
- Defects found should be acknowledged, appreciated as helping the author and handled objectively.

- The review meeting is well managed, so that it is seen as a valuable use of time.
- There is an atmosphere of trust during the review; there is no evaluation of the person, just the work product.
- The use of negative body language can show boredom, exasperation or hostility and should be avoided.
- Ensuring that adequate training is provided, especially for the formal review processes such as inspections.
- The culture of review is all about learning and process improvement and is promoted.

CHECK OF UNDERSTANDING

1. Compare the differences between a walkthrough and an inspection.
2. Name three characteristics of a walkthrough.
3. Identify at least five success factors for a review.

SUMMARY

In this chapter, we have looked at how review techniques and static analysis fit within the test process defined in **Chapter 1**. We have understood that a review is a static test – that is it is a test carried out without executing any code (by reading and commenting on any work product such as a requirement specification, a piece of code or a test plan/test case). We have also looked at the different types of review techniques available, such as walkthroughs and inspections, as well as spending time understanding the benefits of reviews themselves.

Reviews vary in formality. The formality governs the amount of structure and documentation that surround the review itself.

To obtain the most benefit from reviews, they should be carried out as early in the project life cycle as possible, preferably as soon as the document to be reviewed has been written and definitely, in the case of work products such as requirement specifications and designs, before any code is written or executed. The roles of the participant reviewers need to be defined and, in the more structured review techniques, written output from reviews is expected.

We have learnt that static analysis is checking the developed software code before it is executed, checking for defects such as unreachable (dead code) and the misuse of development standards. We have also learnt that static analysis is best carried out using tools, which are referenced in **Chapter 6**.

Example examination questions with answers

E1. K1 question
Which of the following is most likely to be examined using static testing?

a. User guides.
b. Defect reports.
c. Test logs.
d. Attendance reports.

E2. K2 question
Which of the following has the typical formal review activities in the correct sequence?

a. Kick-off, initiate, review meeting, planning, follow-up.
b. Kick-off, planning, review meeting, issue communications and analysis, rework.
c. Planning, initiate, individual review, issue communications and analysis, fixing and reporting.
d. Planning, individual preparation, initiate, individual review, follow-up, fixing and reporting.

E3. K2 question
Which of the following statements are true?

i. Defects are likely to be found earlier in the development process by using reviews.
ii. Walkthroughs require code but static analysis does not require code.
iii. Informal reviews are used to detect potential defects.
iv. Ad hoc techniques need lots of preparation time.
v. Dynamic testing occurs after reviews have been used to find defects.

a. i, ii, iv.
b. ii, iii, v.
c. i, iv, v.
d. i, iii, v.

E4. K2 question
Which of the following defects could be identified by static testing?

a. Execution defects, coding defects and security vulnerabilities.
b. Coding defects, requirements defects and security vulnerabilities.
c. Security vulnerabilities, test basis issues and environment defects.
d. Design defects, user defects and test basis issues.

E5. K1 question
Which *one* of the following roles is typically used in a review?

a. Champion.
b. Author.
c. Project sponsor.
d. Custodian.

E6. K2 question
Which of the following is a success factor for reviews?

a. The total count of lines of code.
b. Walkthrough of a requirements document.
c. Large documents reviewed as a whole.
d. Each review has clear objectives.

Answers to questions in the chapter

SA1. The correct answer is a.

SA2. The correct answer is c.

SA3. The correct answer is d.

Answers to example examination questions

E1. The correct answer is a.
The other answers could be examined in a static review; only a is identified in the syllabus.

E2. The correct answer is c.
The correct sequence is: planning, initiate, individual review, issue communications and analysis, fixing and reporting. All of the other options have either the activities in the wrong order or activities missing from the strict flow.

E3. The correct answer is d.
The other answers are incorrect because:

ii. Walkthroughs do not require code and static analysis does require code.
iv. Ad hoc reviews need little preparation.

E4. The correct answer is b.
All other options have a dynamic testing defect amongst the options.

E5. The correct answer is b.
The Author is the only role that is typically used in a review. A Champion might sponsor the review process but is not a defined role within an actual review; a Project sponsor, if technically competent, might be asked to play a defined role within the review process, but while in that role they will not be a Project sponsor; finally, a Custodian might ensure that the results are stored safely but is not involved in the actual review itself.

E6. The correct answer is d.
Each review has clear objectives. The remaining answers are not success factors for reviews.

4 TEST TECHNIQUES

Brian Hambling

INTRODUCTION

This chapter addresses the area of test techniques, which is a range of techniques used throughout test analysis and design. The 2018 syllabus has significantly reduced the content of this section, so the main flow of the chapter will be aligned with the 2018 syllabus. The more advanced material on white-box techniques has been clearly identified so that those readers interested in understanding how white-box tests are designed can engage with this section, and there are exercises on white-box techniques and on determining the level of test coverage achieved by a suite of tests included with this material. Readers interested only in examinable material can safely skip this material.

The chapter begins with an introduction to key terms and the basic process of creating test suites for execution and then explores the three main categories of test techniques: black-box (also called behavioural or behaviour-based techniques and formerly known as specification-based techniques); white-box (also called structural or structure-based techniques); and experience-based techniques. In each case, specific techniques are explained and examples are given of their use.

In the section on white-box techniques we introduce the theory and skills needed to analyse structured representations (such as control flow graphs and program code) and draw both flow charts and control flow graphs to represent structure and facilitate analysis. Flow charts and control flow graphs have many applications outside software testing and are valuable tools for the simple expression of logical structures such as user interfaces. This section is not essential reading for those aiming only to pass the Foundation Level examination, but will be a sound basis for the more advanced techniques. The rest of the section on white-box techniques is examinable.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions that follow the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as

a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

Categories of test techniques (K2)

- FL-4.1.1 Explain the characteristics, commonalities, and differences between black-box test techniques, white-box test techniques, and experience-based test techniques.

Black-box test techniques (K3)

- FL-4.2.1 Apply equivalence partitioning to derive test cases from given requirements.
- FL-4.2.2 Apply boundary value analysis to derive test cases from given requirements.
- FL-4.2.3 Apply decision table testing to derive test cases from given requirements.
- FL-4.2.4 Apply state transition testing to derive test cases from given requirements.
- FL-4.2.5 Explain how to derive test cases from a use case. (K2)

White-box test techniques (K2)

- FL-4.3.1 Explain statement coverage.
- FL-4.3.2 Explain decision coverage.
- FL-4.3.3 Explain the value of statement and decision coverage.

Experience-based test techniques (K2)

- FL-4.4.1 Explain error guessing.
- FL-4.4.2 Explain exploratory testing.
- FL-4.4.3 Explain checklist-based testing.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter. If you struggled with the questions it suggests that, while your recall of key ideas might be reasonable, your ability to apply the ideas needs developing. You need to study this chapter carefully and be careful to recognise all the connections between individual topics.

Question SA1 (K2)
Which of the following are characteristics of white-box testing?

a. The test basis typically includes specifications and use cases.
b. Test coverage cannot be measured.
c. The test basis can be based on the knowledge and experience of stakeholders.
d. Test coverage is based on the items tested within a given structure.

Question SA2 (K2)
Which of the following statements about statement and decision coverage is true?

a. Statement coverage measures how many statements there are in a fragment of code.
b. Achievement of 100% statement coverage guarantees the achievement of at least 50% of decision coverage.
c. Achievement of 100% decision coverage does not guarantee the achievement of 100% statement coverage.
d. The achievement of 100% decision coverage ensures that the true and false outcomes of every decision in the code are exercised.

Question SA3 (K2)
Which of the following statements correctly characterises use case testing?

a. Use case coverage is measured by the number of error cases tested.
b. Use case coverage is measured as the number of use case behaviours tested divided by the total number of use case behaviours.
c. In use case testing, tests are derived from the basic behaviour, including error cases, defined by a set of use cases, but do not test exceptional or alternative behaviours.
d. In use case testing, tests are derived from the basic, exceptional or alternative behaviours defined by a set of use cases but do not test error cases.

THE TEST DEVELOPMENT PROCESS

The specification of test cases is a key step in any test process. The terms specification and design are used interchangeably in this context; in this section, we discuss the creation of test cases by design.

The design of tests comprises three main steps:

1. identify test conditions;
2. specify test cases;
3. specify test procedures.

Our first task is to become familiar with the terminology.

> A test basis is the body of knowledge used as the basis for test analysis and design.

> A test condition is an aspect of the test basis that is relevant in order to achieve specific test objectives.

In other words, a test condition is some characteristic of our software that we can check with a test or a set of tests.

> A test case is a set of preconditions, inputs, actions (where applicable), expected results and postconditions, based on test conditions.

In other words, a test case: gets the system to some starting point for the test (execution preconditions); then applies a set of input values that should achieve a given outcome (expected result); then leaves the system at some end-point (execution postcondition).

Our test design activity will generate the set of input values and we will predict the expected outcome by, for example, identifying from the specification what should happen when those input values are applied.

We have to define what state the system is in when we start so that it is ready to receive the inputs, and we have to decide what state it is in after the test so that we can check that it ends up in the right place (and to make it easier to design additional test cases that start from where this one ended).

> A test procedure is a sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap-up activities post execution.

A test procedure therefore identifies all the necessary actions in sequence to execute a test. Test procedure specifications are often called test scripts (or sometimes manual test scripts to distinguish them from the automated scripts that control test execution tools, introduced in **Chapter 6**).

So, going back to our three-step process above, we:

1. decide on a test condition, which is typically a small section of the specification for our software under test;
2. design a test case that will verify the test condition;
3. write a test procedure to execute the test; that is, get it into the right starting state, input the values and check the outcome.

Despite the technical language, this is quite a simple set of steps. Of course, we will have to carry out a very large number of these simple steps to test a whole system, but the basic process is still the same. To test a whole system, we write a test execution schedule, which puts all the individual test procedures in the right sequence and sets up the system so that they can be run.

Bear in mind as well that the test development process may be implemented in more or less formal ways. In some situations, it may be appropriate to produce very little documentation and in others a very formal and documented process may be appropriate. It all depends on the context of the testing, taking account of factors such as maturity of development and test processes, the amount of time available and the nature of the system under test. Safety-critical systems, for example, will normally require a more formal test process.

The best way to clarify the process is to work through a simple example.

TEST CASE DESIGN BASICS

Suppose we have a system that contains the following specification for an input screen:

1.2.3 The input screen shall have three fields: a title field with a drop-down selector; a surname field that can accept up to 20 alphabetic characters and the hyphen (-) character; a first name field that can accept up to 20 alphabetic characters. All alphabetic characters shall be case insensitive. All fields must be completed. The data is validated when the Enter key is pressed. If the data is valid, the system moves on to the job input screen; if not, an error message is displayed.

This specification enables us to define test conditions; for example, we could define a test condition for the surname field (i.e. it can accept up to 20 alphabetic characters and the hyphen (-) character) and define a set of test cases to test that field.

To test the surname field, we would have to navigate the system to the appropriate input screen, select a title, tab to the surname field (all this would be setting the test precondition), enter a value (the first part of the set of input values), tab to the first name field and enter a value (the second part of the set of input values that we need because all fields must be completed), then press the Enter key. The system

should either move on to the job input screen (if the data we input was valid) or display an error message (if the input data was not valid). Of course, we would need to test both of these cases.

The preceding paragraph is effectively the test procedure, though we might lay it out differently for real testing.

A good test case needs some extra information. First, it should be traceable back to the test condition and the element of the specification that it is testing; we can do this by applying the specification reference to the test, for example by identifying this test as T1.2.3.1 (because it is the first test associated with specification element 1.2.3).

Second, we would need to add a specific value for the input, say 'Hambling' and 'Brian'. Finally, we would specify that the system should move to the job input screen when 'Enter' is pressed.

TEST CASE DESIGN EXAMPLE

As an example, we could key in the following test cases:

Mr	Hambling	Brian
Ms	Samaroo	Angelina
Ms	Simmonite	Compo
Mr	Hyde-White	Wilfred

All these are valid test cases; even though Compo Simmonite was an imaginary male character in a TV series, the input is correct according to the specification.

We should also test some invalid inputs, such as:

Mr	Thompson1	Geoff
Mr	"Morgan"	Peter
Mr	Williams	'Pete'
Miss	Ruby	Very-long-firstname-rejected

There are many more possibilities that infringe the rules in the specification, but these should serve to illustrate the point. You may be thinking that this simple specification could generate a very large number of test cases – and you would be absolutely right. One of our aims in using systematic test case design techniques will be to cut down the number of tests we need to run to achieve a given level of confidence in the software we are testing.

The test procedure needs to add some details along the following lines:

1. Select the <Name or Personal Details> option from the main menu.
2. Select the 'input' option from the <Name or Personal Details> menu.
3. Select 'Mr' from the 'Title' drop-down menu.
4. Check that the cursor moves to the 'surname' field.
5. Type in 'Hambling' and press the tab key once; check that the cursor moves to the 'first name' field.
6. Type in 'Brian' and press the Enter key.
7. Check that the Job Input screen is displayed.
8. . . .

That should be enough to demonstrate what needs to be defined, and also how slow and tedious such a test is to run, and we have only completed one of the test cases so far!

The test procedure collects together all the test cases related to this specification element so that they can all be executed together as a block; there would be several to test valid and non-valid inputs, as you have seen in the example.

In the wider test process, we move on to the test execution step next. In preparation for execution, the test execution schedule collects together all the tests and sequences them, considering any priorities (highest priority tests are run first) and any dependencies between tests. For example, it makes sense to do all the tests on the input screen together and to do all the tests that use input data afterwards; that way we get the input screen tests to do the data entry that we will need for the later tests. There might also be technical reasons why we run tests in a particular sequence; for example, a test of the password security needs to be done at the beginning of a sequence of tests because we need to be able to get into the system to run the other tests.

THE IDEA OF TEST COVERAGE

Test coverage is a very important idea because it provides a quantitative assessment of the extent and quality of testing. In other words, it answers the question 'How much testing have you done?' in a way that is not open to interpretation. Statements such as 'I'm nearly finished', or 'I've done two weeks' testing' or 'I've done everything in the test plan' generate more questions than they answer. They are statements about how much testing has been done or how much effort has been applied to testing, rather than statements about how effective the testing has been or what has been achieved. We need to know about test coverage for two very important reasons:

- It provides a quantitative measure of the quality of the testing that has been done by measuring what has been achieved.

- It provides a way of estimating how much more testing needs to be done. Using quantitative measures, we can set targets for test coverage and measure progress against them.

Statements like 'I have tested 75 per cent of the decisions' or 'I've tested 80 per cent of the requirements' provide useful information. They are neither subjective nor qualitative; they provide a real measure of what has actually been tested. If we apply coverage measures to testing based on priorities, which are themselves based on the risks addressed by individual tests, we will have a reliable, objective and quantified framework for testing.

Test coverage can be applied to any systematic technique; in this chapter we will apply it to specification-based techniques to measure how much of the functionality has been tested, and to structure-based techniques to measure how much of the code has been tested. Coverage measures may be part of the completion criteria defined in the test plan (step 1 of a generalised test process) and used to determine when to stop testing in the final step of this process.

CHECK OF UNDERSTANDING

1. What defines the process of test execution?
2. Briefly compare a test case and a test condition.
3. Which document identifies the sequence in which tests are executed?
4. Describe the purpose of a test coverage measure.

CATEGORIES OF TEST CASE DESIGN TECHNIQUES

There are very many ways to design test cases. Some are general, others are very specific. Some are very simple to implement, others are difficult and complex to implement. The many excellent books published on software testing techniques every year testify to the rate of development of new and interesting approaches to the challenges that confront the professional software tester.

There is, however, a collection of test case design techniques that has come to be recognised as the most important ones for a tester to learn to apply, and these have been selected as the representatives of test case design for the Foundation Certificate, and hence for this book.

The test case design techniques we will look at are grouped into three categories:

- Those based on deriving test cases directly from a specification or a model of a system or proposed system, known as black-box techniques. So black-box techniques are based on an analysis of the test basis documentation, including

both functional and non-functional aspects. They do not use any information regarding the internal structure of the component or system under test.

- Those based on deriving test cases directly from the structure of a component or system are known as white-box techniques. We will briefly introduce this category, which concentrates on tests based on the code written to implement a component or system in this chapter, but other aspects of structure, such as a menu structure, can be tested in a similar way. An optional section of the chapter provides a more in-depth treatment of these techniques.
- Those based on deriving test cases from the tester's experience of similar systems and general experience of testing, known as experience-based techniques.

It is convenient to categorise techniques for test case design in this way (it is easier for you to remember, for one thing) but do not assume that these are the only categories or the only techniques; there are many more that can be added to the tester's 'tool kit' over time.

The category known as 'black-box' techniques is so-called because the techniques in it take a view of the system that does not need to know what is going on 'inside the box'. Some will recognise 'black box' as the name of anything technical that you can use but about which you know nothing or next to nothing. The natural alternative to 'black box' is 'white box' and so 'white-box' techniques are those that are based on internal structure rather than external function.

Experience-based testing was not really treated as 'proper' testing in testing prehistory, so it was given disdainful names such as 'ad hoc'; the implication that this was not a systematic approach was enough to exclude it from many discussions about testing. Both the intellectual climate and the sophistication of experience-based techniques have moved on from those early days. It is worth bearing in mind that many systems are still tested in an experience-based way, partly because the systems are not specified in enough detail or in a sufficiently structured way to enable other categories of technique to be applied, or because neither the development team nor the testing team have been trained in the use of specification-based or structure-based techniques.

Before we look at these categories in detail, think for a moment about what we are trying to achieve. We want to try to check that a system does everything that its specification says it should do and nothing else. In practice, the 'nothing else' is the hardest part and generates the most tests; that is because there are far more ways of getting something wrong than there are ways of getting it right. Even if we just concentrate on testing that the system does what it is supposed to do, we will still generate a very large number of tests. This will be expensive and time-consuming, which means it probably will not happen, so we need to ensure that our testing is as efficient as possible. As you will see, the best techniques do this by creating the smallest set of tests that will achieve a given objective, and they do that by taking advantage of certain things we have learned about testing; for example, that defects tend to cluster in interesting ways.

Bear this in mind as we take a closer look at the categories of test case design techniques.

CHOOSING TEST TECHNIQUES

The decision of which test technique to use is not a simple one. There are many factors to bear in mind, some of which are listed in the box.

KEY SELECTION FACTORS

- Type of system or component that we are testing; for example it may be a database or a system to control a manufacturing process.
- Any regulatory standards that may apply. These may be related to the type of system; for example safety critical, the type of industry in which the system will be used; for example railway systems, or other regulated applications.
- Customer or contractual requirements. Some customers may demand more rigorous testing than others and this will likely be identified in any contract for purchase of the system.
- Level of risk. Risk may be defined rigorously, as in safety-critical systems, or more colloquially; for example relating to business risk.
- Type of risk. Commercial risk and risk to human safety are examples of different types of risk.
- Test objectives, which will define, formally or informally, how much and what kind of testing is needed.
- Documentation available as a test basis.
- Knowledge of the testers, which will determine what kinds of testing can be designed and implemented.
- Time and budget. Both set limits on how much testing can be done.
- Development life cycle, which may define specific testing stages, as in the V model (see **Chapter 2**), or testing may be an implicit part of development, as in Agile development.
- Use case models. Use case models identify how a system is expected to function from a user perspective, so the use case model will effectively determine the scope and level of testing.
- Experience of the type of defects found in similar systems, which may enable better informed testing that focuses on problem areas.

One or more of these factors may be important on any given occasion. Some leave no room for selection: regulatory or contractual requirements leave the tester with no choice. Test objectives, where they relate to exit criteria such as test coverage, may also lead to mandatory techniques. Where documentation is not available, or where time and budget are limited, the use of experience-based techniques may be favoured. All others provide pointers within a broad framework: level and type of risk will push the tester towards a particular approach, where high risk is a good reason for using systematic

techniques; knowledge of testers, especially where this is limited, may narrow down the available choices; the type of system and the development life cycle will encourage testers to lean in one direction or another depending on their own particular experience. There are few clear-cut cases, and the exercise of sound judgement in selecting appropriate techniques is a mark of a good test manager or team leader.

BLACK-BOX TEST TECHNIQUES

The main thing about black-box test techniques is that they derive test cases directly from the specification or from some other kind of model of what the system should do. The source of information on which to base testing is known as the 'test basis'. If a test basis is well defined and adequately structured, we can easily identify test conditions from which test cases can be derived.

The most important point about black-box test techniques is that specifications or models do not (and should not) define how a system should achieve the specified behaviour when it is built; it is a specification of the required (or at least desired) behaviour. One of the hard lessons that software engineers have learned from experience is that it is important to separate the definition of what a system should do (a specification) from the definition of how it should work (a design). This separation allows the two specialist groups (testers for specifications and designers for design) to work independently so that we can later check that they have arrived at the same place; that is, they have together built a system and tested that it works according to its specification.

If we set up test cases so that we check that desired behaviour actually occurs, then we are acting independently of the developers. If they have misunderstood the specification or chosen to change it in some way without telling anyone then their implementation will deliver behaviour that is different from what the model or specification said the system behaviour should be. Our test, based solely on the specification, will therefore fail and we will have uncovered a problem.

Bear in mind that not all systems are defined by a detailed formal specification. In some cases, the model we use may be quite informal. If there is no specification at all, the tester may have to build a model of the proposed system, perhaps by interviewing key stakeholders to understand what their expectations are. However formal or informal the model is, and however it is built, it provides a test basis from which we can generate tests systematically.

Remember, also, that the specification can contain non-functional elements as well as functions; topics such as reliability, usability and performance are examples. These need to be systematically tested as well.

What we need, then, are techniques that can explore the specified behaviour systematically and thoroughly in a way that is as efficient as we can make it. In addition, we use what we know about software to 'home in' on problems; each of the test case design techniques is based on some simple principles that arise from what we know in general about software behaviour.

You need to know five specification-based techniques for the Foundation Certificate:

- equivalence partitioning;
- boundary value analysis;
- decision table testing;
- state transition testing;
- use case testing.

You should be capable of generating test cases for the first four of these techniques.

CHECK OF UNDERSTANDING

1. What do we call the category of test case design techniques that requires knowledge of how the system under test actually works?
2. What do black-box techniques derive their test cases from?
3. How do we make specification-based testing work when there is no specification?

Equivalence partitioning

Input partitions

Equivalence partitioning is based on a very simple idea: it is that in many cases the inputs to a program can be 'chunked' into groups of similar inputs. For example, a program that accepts integer values can accept as valid any input that is an integer (i.e. a whole number) and should reject anything else (such as a real number or a character). The range of integers is infinite, though the computer will limit this to some finite value in both the negative and positive directions (simply because it can only handle numbers of a certain size; it is a finite machine). Let us suppose, for the sake of an example, that the program accepts any value between –10,000 and +10,000 (computers actually represent numbers in binary form, which makes the numbers look much less like the ones we are familiar with, but we will stick to a familiar representation). If we imagine a program that separates numbers into two groups according to whether they are positive or negative, the total range of integers could be split into three 'partitions': the values that are less than zero; zero; and the values that are greater than zero. Each of these is known as an 'equivalence partition' because every value inside the partition is exactly equivalent to any other value as far as our program is concerned. So, if the computer accepts –2,905 as a valid negative integer we expect it also to accept –3. Similarly, if it accepts 100 it should also accept 2,345 as a positive integer. Note that we are treating zero as a special case. We could, if we chose to, include zero with the positive integers, but my rudimentary specification did not specify that clearly, so it is really left as an undefined value (and it is not untypical to find such ambiguities or undefined areas in specifications). It often suits us to treat zero as a special case for testing where ranges of numbers are involved; we treat it as an equivalence partition with only one member. So, we have three valid equivalence partitions in this case.

The equivalence partitioning technique takes advantage of the properties of equivalence partitions to reduce the number of test cases we need to write. Since all the values in an equivalence partition are handled in exactly the same way by a given program, we need only test one of them as a representative of the partition. In the example given, then, we need any positive integer, any negative integer and zero. We generally select values somewhere near the middle of each partition, so we might choose, say, –5,000; 0; and 5,000 as our representatives. These three test inputs exercise all three partitions and the theory tells us that if the program treats these three values correctly, it is very likely to treat all of the other values, all 19,998 of them in this case, correctly.

The partitions we have identified so far are called valid equivalence partitions because they partition the collection of valid inputs, but there are other possible inputs to this program that are not valid – real numbers, for example. We also have two input partitions of integers that are not valid: integers less than –10,000 and integers greater than 10,000. We should test that the program does not accept these, which is just as important as the program accepting valid inputs.

If you think about the example we have been using, you will soon recognise that there are far more possible non-valid inputs than valid ones, since all the real numbers (e.g. numbers containing decimals) and all characters are non-valid in this case. It is generally the case that there are far more ways to provide incorrect input than there are to provide correct input; as a result, we need to ensure that we have tested the program against the possible non-valid inputs. Here again, equivalence partitioning comes to our aid: all real numbers are equally non-valid, as are all alphabetic characters. These represent two non-valid partitions that we should test, using values such as 9.45 and 'r' respectively. There will be many other possible non-valid input partitions, so we may have to limit the test cases to the ones that are most likely to crop up in a real situation.

EXAMPLE EQUIVALENCE PARTITIONS

- valid input: integers in the range 100 to 999;
- valid partition: 100 to 999 inclusive;
- non-valid partitions: less than 100, more than 999, real (decimal) numbers and non-numeric characters;
- valid input: names with up to 20 alphabetic characters;
- valid partition: strings of up to 20 alphabetic characters;
- non-valid partitions: strings of more than 20 alphabetic characters, strings containing non-alphabetic characters.

Exercise 4.1

Suppose you have a bank account that offers variable interest rates: 0.5 per cent for the first £1,000 credit; 1 per cent for the next £1,000; 1.5 per cent for the rest. If you wanted to check that the bank was handling your account correctly, what valid input partitions might you use?

The answer can be found at the end of the chapter.

Output partitions

Just as the input to a program can be partitioned, so can the output. The program in the exercise above could produce outputs of 0.5 per cent, 1 per cent and 1.5 per cent, so we can use test cases that generate each of these outputs as an alternative to generating input partitions. An input value in the range £0.00–£1,000.00 generates the 0.5 per cent output; a value in the range £1,001.00–£2,000.00 generates the 1 per cent output; a value greater than £2,000.00 generates the 1.5 per cent output.

Other partitions

If we know enough about an application, we may be able to partition other values instead of or as well as input and output. For example, if a program handles input requests by placing them on one of a number of queues we could, in principle, check that requests end up on the right queue. In this case, a stream of inputs can be partitioned according to the queue we anticipate it will be placed into. This is more technical and difficult than input or output partitioning, but it is an option that can be considered when appropriate.

PARTITIONS – EXAMPLE 4.1

A mail-order company charges £2.95 postage for deliveries if the package weighs less than 2 kg, £3.95 if the package weighs 2 kg or more but less than 5 kg, and £5 for packages weighing 5 kg or more. Generate a set of valid test cases using equivalence partitioning.

The valid input partitions are: under 2 kg; 2 kg or over but less than 5 kg; and 5 kg or over.

Input values could be 1 kg, 3.5 kg, 7.5 kg. These produce expected results of £2.95, £3.95 and £5 respectively.

In this case there are no non-valid inputs (unless the scales fail).

Exercise 4.2

A mail-order company selling flower seeds charges £3.95 for postage and packing on all orders up to £20 value and £4.95 for orders above £20 value and up to £40 value. For orders above £40 value, there is no charge for postage and packing.

If you were using equivalence partitioning to prepare test cases for the postage and packing charges what valid partitions would you define?

What about non-valid partitions?

The answer can be found at the end of the chapter.

Boundary value analysis

One thing we know about the kinds of mistakes that programmers make is that errors tend to cluster around boundaries. For example, if a program should accept a sequence of numbers between 1 and 10, the most likely fault will be that values just outside this range are incorrectly accepted or that values just inside the range are incorrectly rejected. In the programming world, these faults coincide with particular programming structures such as the number of times a program loop is executed or the exact point at which a loop should stop executing.

This works well with our equivalence partitioning idea because partitions must have boundaries. (Where partitions do not have boundaries, for example in the set of days of the week, the boundary analysis technique cannot be used.) A partition of integers between 1 and 99, for instance, has a lowest value, 1, and a highest value, 99. These are called boundary values. Actually, they are called valid boundary values because they are the boundaries on the inside of a valid partition. What about the values on the outside? Yes, they have boundaries too. So, the boundary of the non-valid values at the lower end will be zero because it is the first value you come to when you step outside the partition at the bottom end. (You can also think of this as the highest value inside the non-valid partition of integers that are less than one, of course.) At the top end of the range we also have a non-valid boundary value, 100.

This is the boundary value technique, more or less. For most practical purposes the boundary value analysis technique needs to identify just two values at each boundary. For reasons that need not detain us here there is an alternative version of the technique that uses three values at each boundary. For this variant, which is the one documented in BS 7925-2, we include one more value at each boundary when we use boundary value analysis: the rule is that we use the boundary value itself and one value (as close as you can get) either side of the boundary.

So, in this case lower boundary values will be 0, 1, 2 and upper boundary values will be 98, 99, 100. What does 'as close as we can get' mean? It means take the next value in sequence using the precision that has been applied to the partition. If the numbers are to a precision of 0.01, for example, the lower boundary values are 0.99, 1.00, 1.01 and the upper boundary values are 98.99, 99.00, 99.01.

When you come to take your exam, you will find that the exam recognises that there are two possible approaches to boundary value analysis. For this reason, any questions about boundary value analysis will clearly signal whether you are expected to identify two or three values at any boundary. You will find that this causes no problems, but there are examples below using both the two value and the three value approach, just to be on the safe side, to ensure that you do not get taken by surprise in the exam.

The best way to consolidate the idea of boundaries is to look at some examples.

BOUNDARY VALUES – EXAMPLE 4.2

- The boiling point of water – the boundary is at 100 degrees Celsius, so for the three value boundary approach, the boundary values will be 99 degrees, 100 degrees, 101 degrees – unless you have a very accurate digital thermometer, in which case they could be 99.9 degrees, 100.0 degrees, 100.1 degrees. For the two value approach the corresponding values are 100 and 101.
- Exam pass – if an exam has a pass boundary at 40 per cent, merit at 60 per cent and distinction at 80 per cent the three value boundaries are 39, 40, 41 for pass, 59, 60, 61 for merit, 79, 80, 81 for distinction. It is unlikely that marks would be recorded at any greater precision than whole numbers. The two value equivalents are 39 and 40, 59 and 60, and 79 and 80 respectively.

Exercise 4.3

A system is designed to accept scores from independent markers who have marked the same examination script. Each script should have five individual marks, each of which is out of 20, and a total for the script. Two markers' scores are compared and differences greater than three in any question score or 10 overall are flagged for further examination.

Using equivalence partitioning and boundary value analysis, identify the boundary values that you would explore for this scenario.

(In practice, some of the boundary values might actually be in other equivalence partitions, and we do not need to test them twice, so the total number of boundary values requiring testing might be less than you might expect.) The answer can be found at the end of the chapter.

CHECK OF UNDERSTANDING

1. What is the relationship between a partition and a boundary?
2. Why are equivalence partitioning and boundary value analysis often used together?
3. Explain what is meant by 'as close as possible to a boundary'?

Decision table testing

Specifications often contain business rules to define the functions of the system and the conditions under which each function operates. Individual decisions are usually simple, but the overall effect of these logical conditions can become quite complex. As testers we need to be able to assure ourselves that every combination of these conditions

that might occur has been tested, so we need to capture all the decisions in a way that enables us to explore their combinations. The mechanism usually used to capture the logical decisions is called a decision table.

A decision table lists all the input conditions that can occur and all the actions that can arise from them. These are structured into a table as rows, with the conditions at the top of the table and the possible actions at the bottom. Business rules, which involve combinations of conditions to produce some combination of actions, are arranged across the top. Each column therefore represents a single business rule (or just 'rule') and shows how input conditions combine to produce actions. Thus, each column represents a possible test case, since it identifies both inputs and expected outputs. This is shown schematically in the box below.

DECISION TABLE STRUCTURE

	Business rule 1	Business rule 2	Business rule 3
Condition 1	T	F	T
Condition 2	T	T	T
Condition 3	T	–	F
Action 1	Y	N	Y
Action 2	N	Y	Y

Business rule 1 requires all conditions to be true to generate action 1. Business rule 2 results in action 2 if condition 1 is false and condition 2 is true, but does not depend on condition 3. Business rule 3 requires conditions 1 and 2 to be true and condition 3 to be false.

In reality the number of conditions and actions can be quite large, but usually the number of combinations producing specific actions is relatively small. For this reason, we do not enter every possible combination of conditions into our decision table but restrict it to those combinations that correspond to business rules – this is called a limited entry decision table to distinguish it from a decision table with all combinations of inputs identified. In this chapter we will always mean the limited entry kind when we refer to a decision table.

As usual, we use an example to clarify what we mean.

DECISION TABLE TESTING – EXAMPLE 4.3

A supermarket has a loyalty scheme that is offered to all customers. Loyalty cardholders enjoy the benefits of either additional discounts on all purchases (rule 3) or the acquisition of loyalty points (rule 4), which can be converted into vouchers for the supermarket or to equivalent points in schemes run by partners. Customers without a loyalty card receive an additional discount only if they spend more than £100 on any one visit to the store (rule 2), otherwise only the special offers offered to all customers apply (rule 1).

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions: Customer without loyalty card	T	T	F	F
Customer with loyalty card	F	F	T	T
Extra discount selected	–	–	T	F
Spend > £100	F	T	–	–
Actions: No discount	T	F	F	F
Extra discount	F	T	T	F
Loyalty points	F	F	F	T

From the decision table, we can determine test cases by setting values for the conditions and determining the expected output; for example, from rule 1 we could input a normal customer with a £50 transaction and check that no discount was applied. The same customer with a £150 transaction (rule 2) should attract a discount. Thus, we can see that each column of the decision table represents a possible test case.

CHECK OF UNDERSTANDING

1. What is a decision table derived from?
2. Why does decision table testing use limited entry decision tables?
3. Describe how test cases are identified from decision tables.
4. Which element of a decision table defines the expected output for a test case?

Exercise 4.4

A mutual insurance company has decided to float its shares on the stock exchange and is offering its members rewards for their past custom at the time of flotation. Anyone with a current policy will benefit provided it is a 'with-profits' policy and they have held it since 2001. Those who meet these criteria can opt for either a cash payment or an allocation of shares in the new company; those who have held a qualifying policy for less than the required time will be eligible for a cash payment but not for shares. Here is a decision table reflecting those rules.

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**
Conditions: Current policy holder	T	T	F	F
Policy holder since 2001	N	Y	N	–
'With-profits' policy	Y	Y	N	–
Actions: Eligible for cash payment	Y	Y	N	N
Eligible for share allocations	N	Y	N	N

What result would you expect to get for the following test case?

Billy Bunter is a current policy holder who has held a 'with-profits' policy since 2003.

The answer can be found at the end of the chapter.

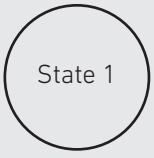
State transition testing

The previous technique, decision table testing, is particularly useful in systems where combinations of input conditions produce various actions. Now we consider a similar technique, but this time we are concerned with systems in which outputs are triggered by changes to the input conditions, or changes of 'state'; in other words, behaviour depends on current state and past state, and it is the transitions that trigger system behaviour. It will be no surprise to learn that this technique is known as state transition testing or that the main diagram used in the technique is called a state transition diagram.

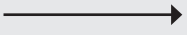
Look at the box to see an example of a state transition diagram.

STATE TRANSITION DIAGRAMS

A state transition diagram is a representation of the behaviour of a system. It is made up from just two symbols.



which is the symbol for a state. A state is just what it says it is: the system is 'static', in a stable condition from which it will only change if it is stimulated by an event of some kind. For example, a TV stays 'on' unless you turn it 'off'; a multifunction watch tells the time unless you change mode. The second is



which is the symbol for a transition; that is, a change from one state to another. The state change will be triggered by an event (e.g. pressing a button or switching a switch). The transition will be labelled with the event that caused it and any action that arises from it. So, we might have 'mode button pressed' as an event and 'presentation changes' as the action. Usually (but not necessarily) the start state will have a double arrowhead pointing to it. Often the start state is obvious anyway.

If we have a state transition diagram representation of a system, we can analyse the behaviour in terms of what happens when a transition occurs.

Transitions are caused by events and they may generate outputs and/or changes of state. An event is anything that acts as a trigger for a change; it could be an input to the system, or it could be something inside the system that changes for some reason, such as a database field being updated.

In some cases, an event generates an output; in others, the event changes the system's internal state without generating an output; and in still others, an event may cause an output and a change of state. What happens for each change is always deducible from the state transition diagram.

STATE TRANSITION DIAGRAM – EXAMPLE 4.4

A hill-walker's watch has two modes: Time and Altimeter. In Time mode, pressing the Mode switch causes the watch to switch to Alt mode; pressing Mode again returns to Time mode. While the watch is in Alt mode the Set button has no effect.

When the watch is in Time mode pressing the Set button transitions the watch into Set Hrs, from which the Hrs display can be incremented by pressing the Set

button. If the Mode switch is pressed while the watch is in Set Hrs mode the watch transitions to Set Mins mode, in which pressing the Set button increments the Mins display. If the Mode button is pressed in this mode, the watch transitions back to Time mode (Figure 4.1).

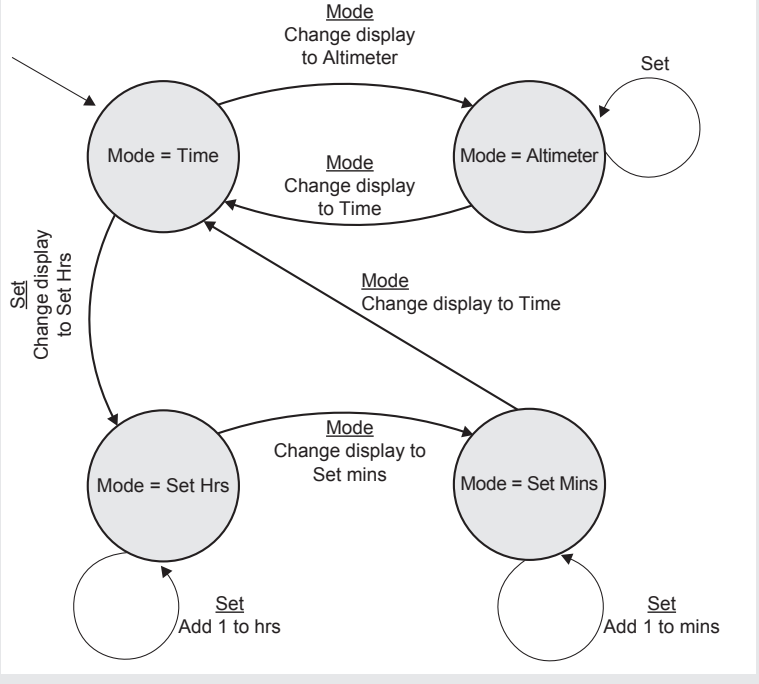
Note that not all events have an effect in all states. Where an event does not have an effect on a given state it is usually omitted, but it can be shown as an arrow starting from the state and returning to the same state to indicate that no transition takes place; this is sometimes known as a 'null' transition or an 'invalid' transition.

Rather than work out what happens for each event each time we want to initiate a test, we can take the intermediate step of creating what is known as a state table (ST). An ST records all the possible events and all the possible states; for each combination of event and state it shows the outcome in terms of the new state and any outputs that are generated.

The ST is the source from which we usually derive test cases. It makes sense to do it this way because the analysis of state transitions takes time and can be a source of errors; it is better to do this task once and then have a simple way of generating tests from it than to do it every time we want to generate a new test case.

Table 4.1 provides an example of what an ST looks like.

Figure 4.1 State transition diagram of the hill-walker's watch



STATE TABLE – EXAMPLE 4.4

An ST has a row for each state in the state transition diagram and a column for every event. For a given row and column intersection, we read off the state from the state transition diagram and note what effect (if any) each event has. If the event has no effect, we label the table entry with a symbol that indicates that nothing happens; this is sometimes called a 'null' transition or an 'invalid' transition. If the event does have an effect, label the table entry with the state to which the system transitions when the given event occurs; if there is also an output (there is sometimes but not always) the output is indicated in the same table entry separated from the new state by the '/' symbol. The example shown in Table 4.1 is the ST for Figure 4.1, which we drew in the previous box.

Table 4.1 ST for the hill-walker's watch

	Mode	Set
Mode = Time	Mode = Altimeter/Change display to Altimeter	Mode = Set Hrs/Change display to Set Hrs
Mode = Altimeter	Mode = Time/Change display to Time	Null
Set Hrs	Mode = Set Mins/Change display to Set Mins	Set Hrs/Add 1 to Hrs
Set Mins	Mode = Time/Change display to Time	Set Mins/Add 1 to Mins

Once we have an ST it is a simple exercise to generate the test cases that we need to exercise the functionality by triggering state changes.

STATE TRANSITION TESTING – EXAMPLE 4.4

We generate test cases by stepping through the ST. If we begin in Time mode, then the first test case might be to press Mode and observe that the watch changes to Alt state; pressing Mode again becomes test case 2, which returns the watch to Time state. Test case 3 could press Set and observe the change to Set Hrs mode and then try a number of presses of Set to check that the incrementing mechanism works. In this way we can work our way systematically round the ST until every single transition has been exercised. If we want to be more sophisticated, we can exercise pairs of transitions; for example, pressing Set twice as a single test case to check that Hrs increments correctly. We should also test all the negative cases; that is, those cases where the ST indicates there is no valid transition.

CHECK OF UNDERSTANDING

1. What is the main use of an ST for testers?
2. Name three components of a state transition diagram.
3. How are negative tests identified from an ST?
4. What is meant by the term 'invalid' transition?

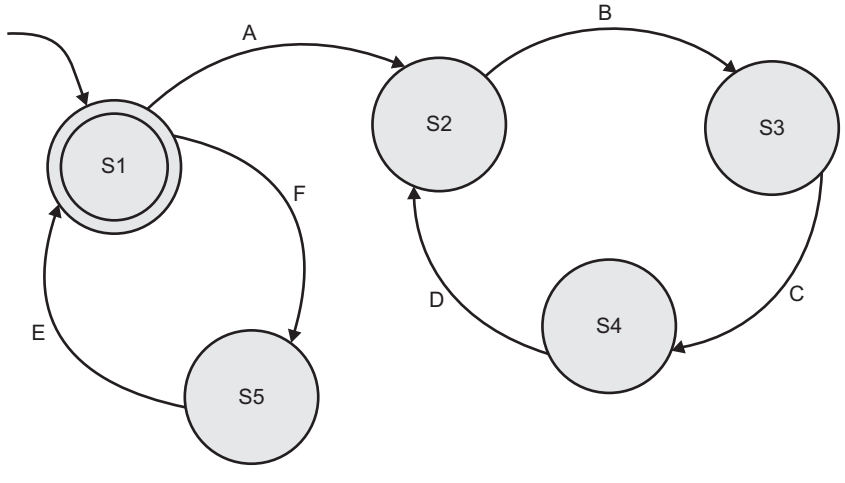
Exercise 4.5

In the state transition diagram in Figure 4.2, which of the sequences of transitions below is valid?

a. ABCDE
b. FEABC
c. ABCEF
d. EFADC

The answer can be found at the end of the chapter.

Figure 4.2 State transition diagram



Use case testing

Use cases are one way of specifying functionality as business scenarios or process flows. They capture the individual interactions between 'actors' and the system. An actor represents a particular type of user and the use cases capture the interactions that each user takes part in to produce some output that is of value. Test cases based on use cases at the business process level, often called scenarios, are particularly useful in exercising

business rules or process flows and will often identify gaps or weaknesses in these that would not be found by exercising individual components in isolation. This makes use case testing very effective in defining acceptance tests because the use cases represent actual likely use.

Use cases may also be defined at the system level, with preconditions that define the state the system needs to be in on entry to a use case to enable the use case to complete successfully, and postconditions that define the state of the system on completion of the use case. Use cases typically have a mainstream path, defining the expected behaviour, and one or more alternative paths related to such aspects as error conditions. Well-defined use cases can therefore be an excellent basis for system level testing, and they can also help to uncover integration defects caused by incorrect interaction or communication between components.

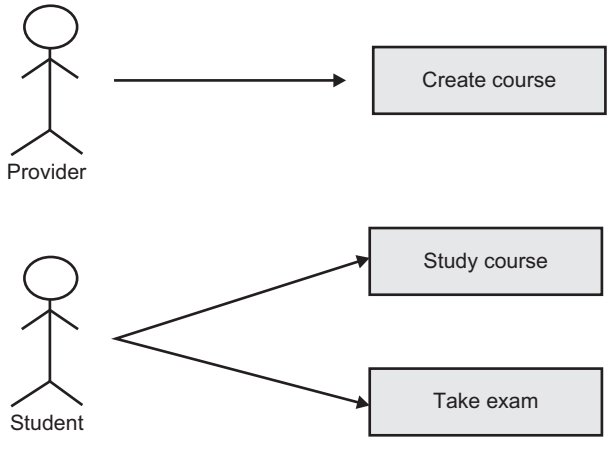
In practice, writing a test case to represent each use case is often a good starting point for testing, and use case testing can be combined with other specification-based testing.

USE CASES

In a use case diagram (e.g. Figure 4.3), each type of user is known as an actor and an actor stands for all users of the type. Use cases are activities carried out for that actor by the system. This is, in effect, a high-level view of requirements.

The diagram alone does not provide enough detail for testing, so we need some textual description of the processes involved as well.

Figure 4.3 Use case example



Use case testing has the major benefit that it relates to real user processes, so it offers an opportunity to exercise a complete process flow. The principles applied elsewhere can be applied here: first, test the highest priority (highest value) use cases by taking typical examples; then exercise some attempts at incorrect process flows; and then exercise the boundaries.

CHECK OF UNDERSTANDING

1. What is the purpose of a use case?
2. What is the relationship between a use case and a test case?
3. Briefly compare equivalence partitioning and use case testing.

WHITE-BOX TEST TECHNIQUES

White-box test techniques are used to explore system or component structures at several levels. At the component level, the structures of interest will be program structures such as program statements and decisions; at the integration level, we may be interested in exploring the way components interact with other components (in what is usually termed a calling structure); at the system level, we may be interested in how users will interact with a menu structure. All these are examples of structures and all may be tested using white-box test techniques.

Instead of exercising a component or system to see if it functions correctly, using a specification or model of the system to determine correct functioning, white-box tests focus on ensuring that elements of the structure of a component, sub-system or system are correctly exercised. For example, we can use white-box test techniques to ensure that each statement in the code of a component is executed at least once, using a technique called statement testing; we can test that each decision in a component is exercised and behaves as expected; we can test that components interact correctly in a sub-system; we can test that user interaction with menus is correct.

We will focus here on statement testing and decision testing, both of which are usually deployed at the component or program level.

Statement testing and coverage

Statement testing exercises the executable statements in the code. Tests are designed to attempt to force the program to execute particular statements by setting a program to a known start state and inputting data that will cause the required statements to be executed. The expected outputs of the test will also be predicted from the program code so that, on completion of the test, it can be determined whether the test executed the required statements. An analyser tool can also be used to determine which statements were executed by a test run.

Statement coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage. A target for statement coverage can be set and statement testing would continue until the required target statement coverage is achieved.

Decision testing and coverage

Decision testing exercises the decisions in the code. In a similar way to statement testing, tests are designed to attempt to force the program to execute particular decisions in particular ways. This will be achieved by setting the program to a known start state and inputting data that is expected to set given decisions so that the program exits them from the required exit (true or false). To do this, the test cases follow the control flows that occur from a decision point (e.g. for an IF statement, one for the true outcome and one for the false outcome; for a CASE statement, test cases are required for all the possible outcomes, including the default outcome). As with statement testing, the expected outputs of the test will also be predicted from the program code so that, on completion of the test, it can be determined whether the test executed the required decisions appropriately. An analyser tool can also be used to determine which decisions were executed by a test run.

Decision coverage is measured as the number of decision outcomes executed by the tests divided by the total number of decision outcomes in the test object, normally expressed as a percentage. A target for decision coverage can be set and decision testing would continue until the required target decision coverage is achieved.

The value of statement and decision testing

When 100 per cent statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Of the two white-box techniques discussed in this syllabus, statement testing may provide less coverage than decision testing. To achieve 100 per cent decision coverage, all decision outcomes must be exercised, so for each decision, both the true outcome and the false outcome, even when there is no explicit false statement (e.g. in the case of an IF statement without an ELSE in the code), must be executed successfully.

Statement coverage can help to find defects in code that was not exercised by other tests. For example, a functional test may run and confirm the correct output, even though the program does not always process the inputs correctly. A functional test is limited to checking that a given input achieves the correct functional output, but it may be that it only exercises part of a program. A more thorough test of the program in isolation may discover that the program does not always behave exactly as expected, and this defect may have functional implications that were not discovered during black-box testing.

Similarly, decision testing may help to find defects in code where functional tests have not forced the program to take both true and false decision exits. Here again, a more thorough exploration of the decision structures in the code can unearth problems that could not easily be detected, or could not be detected at all, by black-box testing.

Achieving 100 per cent decision coverage guarantees that 100 per cent statement coverage has also been achieved, but the reverse is not necessarily true.

The following material is not examinable. Readers who wish to skip this material at first reading may resume the examinable material on page 145.

WHITE-BOX TESTING IN DETAIL

White-box test techniques are used to explore system or component structures at several levels. At the component level, the structures of interest are program structures such as decisions; at the integration level we may be interested in exploring the way components interact with other components (in what is usually termed a calling structure); at the system level, we may be interested in how users will interact with a menu structure. All these are examples of structures and all may be tested using white-box test case design techniques. Instead of exercising a component or system to see if it functions correctly, white-box tests focus on ensuring that particular elements of the structure itself are correctly exercised. For example, we can use structural testing techniques to ensure that each statement in the code of a component is executed at least once. At the component level, where white-box testing is most commonly used, the test case design techniques involve generating test cases from code, so we need to be able to read and analyse code. As you will see later, in **Chapter 6**, code analysis and white-box testing at the component level are mostly done by specialist tools, but a knowledge of the techniques is still valuable. You may wish to run simple test cases on code to ensure that it is basically sound before you begin detailed functional testing, or you may want to interpret test results from programmers to ensure that their testing adequately exercises the code.

The CTFL syllabus does not require this complete level of understanding and takes a simplified approach that uses simplified control flow graphs (introduced in the section on **Simplified control flow graphs**). The following paragraphs can therefore be considered optional for readers whose interest is solely to pass the CTFL exam at this stage but, for anyone intending to progress to the more advanced levels, they provide an essential introduction to code analysis that will be required for these exams and that will be needed in testing real software.

Our starting point, then, is the code itself. The best route to a full understanding of white-box techniques is to start here and work through the whole section.

READING AND INTERPRETING CODE

Often the term 'code' will be taken to mean pseudo code, and this is particularly relevant when addressing a non-specific audience or an international audience. Pseudo code is a much more limited language than any real programming language, but it enables designers to create all the main control structures needed by programs. It is sometimes used to document designs before they are coded into a programming language.

In the next few boxes we will introduce all the essential elements of pseudo code that you will need to be able to analyse code and create test cases.

Wherever you see the word 'code' from here on in this chapter, read it as 'pseudo code'.

Real programming languages have a wide variety of forms and structures – so many that we could not adequately cover them all. The advantage of pseudo code in this respect is that it has a simple structure.

OVERALL PROGRAM STRUCTURE

Code can be of two types: executable and non-executable. Executable code instructs the computer to take some action; non-executable code is used to prepare the computer to do its calculations, but it does not involve any actions. For example, reserving space to store a calculation (this is called a declaration statement) involves no actions. In pseudo code, non-executable statements will be at the beginning of the program; the start of the executable part is usually identified by BEGIN, and the end of the program by END. So, we get the following structure:

```
1   Non-executable statements
2 BEGIN
3
4   Executable statements
5
6 END
```

If we were counting executable statements, we would count lines 2, 4 and 6. Line 1 is not counted because it is non-executable. Lines 3 and 5 are ignored because they are blank.

If there are no non-executable statements there may be no BEGIN or END either, but there will always be something separating non-executable from executable statements where both are present.

Now we have a picture of an overall program structure we can look inside the code. Surprisingly, there are only three ways that executable code can be structured, so we only have three structures to learn. The first is simple and is known as sequence: that just means that the statements are exercised one after the other as they appear on the page. The second structure is called selection: in this case the computer has to decide if a condition (known as a Boolean condition) is true or false. If it is true the computer takes one route, and if it is false the computer takes a different route. Selection structures therefore involve decisions. The third structure is called iteration: it simply involves the computer exercising a chunk of code more than once; the number of times it exercises the chunk of code depends on the value of a condition (just as in the selection case). Let us look at that a little closer.

PROGRAMMING STRUCTURES

SEQUENCE

The following program is purely sequential:

```
1      Read A
2      Read B
3      C = A + B
```

The BEGIN and END have been omitted in this case, since there were no non-executable statements; this is not strictly correct but is common practice, so it is wise to be aware of it and remember to check whether there are any non-executable statements when you do see BEGIN and END in a program. The computer would execute those three statements in sequence, so it would read (input) a value into A (this is just a name for a storage location), then read another value into B, and finally add them together and put the answer into C.

SELECTION

```
1      IF P > 3
2      THEN
3      X = X + Y
4      ELSE
5      X = X - Y
6      ENDIF
```

Here we ask the computer to evaluate the condition P > 3, which means compare the value that is in location P with 3. If the value in P is greater than 3, then the condition is true; if not, the condition is false. The computer then selects which statement to execute next. If the condition is true it will execute the part labelled THEN, so it executes line 3. If the condition is false, it will execute line 5. After it has executed either line 3 or line 5 it will go to line 6, which is the end of the selection (IF THEN ELSE) structure. From there, it will continue with the next line in sequence.

There may not always be an ELSE part, as below:

```
1      IF P > 3
2      THEN
3      X = X + Y
4      ENDIF
```

In this case the computer executes line 3 if the condition is true or moves on to line 4 (the next line in sequence) if the condition is false.

ITERATION

Iteration structures are called loops. The most common loop is known as a DO WHILE (or WHILE DO) loop and is illustrated below:

```
1      X = 15
2      Count = 0
3      WHILE X < 20 DO
4      X = X + 1
5      Count = Count + 1
6      END DO
```

As with the selection structures, there is a decision. In this case the condition that is tested at the decision is X < 20. If the condition is true, the program 'enters the loop' by executing the code between DO and END DO. In this case the value of X is increased by one and the value of Count is increased by one. When this is done the program goes back to line 3 and repeats the test. If X < 20 is still true, the program 'enters the loop' again. This continues as long as the condition is true. If the condition is false, the program goes directly to line 6 and then continues to the next sequential instruction. In the program fragment above, the loop will be executed five times before the value of X reaches 20 and causes the loop to terminate. The value of Count will then be 5.

There is another variation of the loop structure known as a REPEAT UNTIL loop. It looks like this:

```
1      X = 15
2      Count = 0
3      REPEAT
4      X = X + 1
5      Count = Count + 1
6      UNTIL X = 20
```

The difference from a DO WHILE loop is that the condition is at the end, so the loop will always be executed at least once. Every time the code inside the loop is executed, the program checks the condition. When the condition is true, the program continues with the next sequential instruction. The outcome of this REPEAT UNTIL loop will be exactly the same as the DO WHILE loop above.

CHECK OF UNDERSTANDING

1. What is meant by the term executable statement?
2. Briefly describe the two forms of looping structure introduced in this section.
3. What is a selection structure?
4. How many different paths are there through a selection structure?

Flow charts

Now that we can read code, we can go a step further and create a visual representation of the structure that is much easier to work with. The simplest visual structure to draw is the flow chart, which has only two symbols. Rectangles represent sequential

statements and diamonds represent decisions. More than one sequential statement can be placed inside a single rectangle as long as there are no decisions in the sequence. Any decision is represented by a diamond, including those associated with loops.

Let us look at our earlier examples again.

To create a flow chart representation of a complete program, all we need to do is to connect together all the different bits of structure.

Figure 4.4 Flow chart for a sequential program

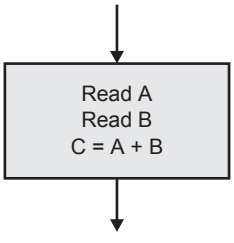


Figure 4.5 Flow chart for a selection (decision) structure

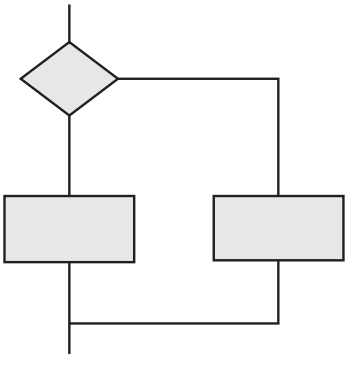
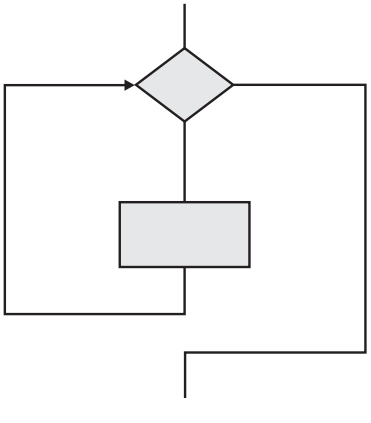


Figure 4.6 Flow chart for an iteration (loop) structure



PROGRAM ANALYSIS – EXAMPLE 4.5

Here is a simple program for calculating the mean and maximum of three integers.

```
1 Program MaxandMean
2
3     A, B, C, Maximum: Integer
4     Mean: Real
5
6 Begin
7
8     Read A
9     Read B
10    Read C
11    Mean = (A + B + C)/3
12
13    If A > B
14    Then
15    If A > C
16    Then
17    Maximum = A
18    Else
19    Maximum = C
20    Endif
21    Else
22    If B > C
23    Then
24    Maximum = B
25    Else
26    Maximum = C
27    Endif
28    Endif
29
30    Print ("Mean of A, B and C is ", Mean)
31    Print ("Maximum of A, B, C is ", Maximum)
32
33 End
```

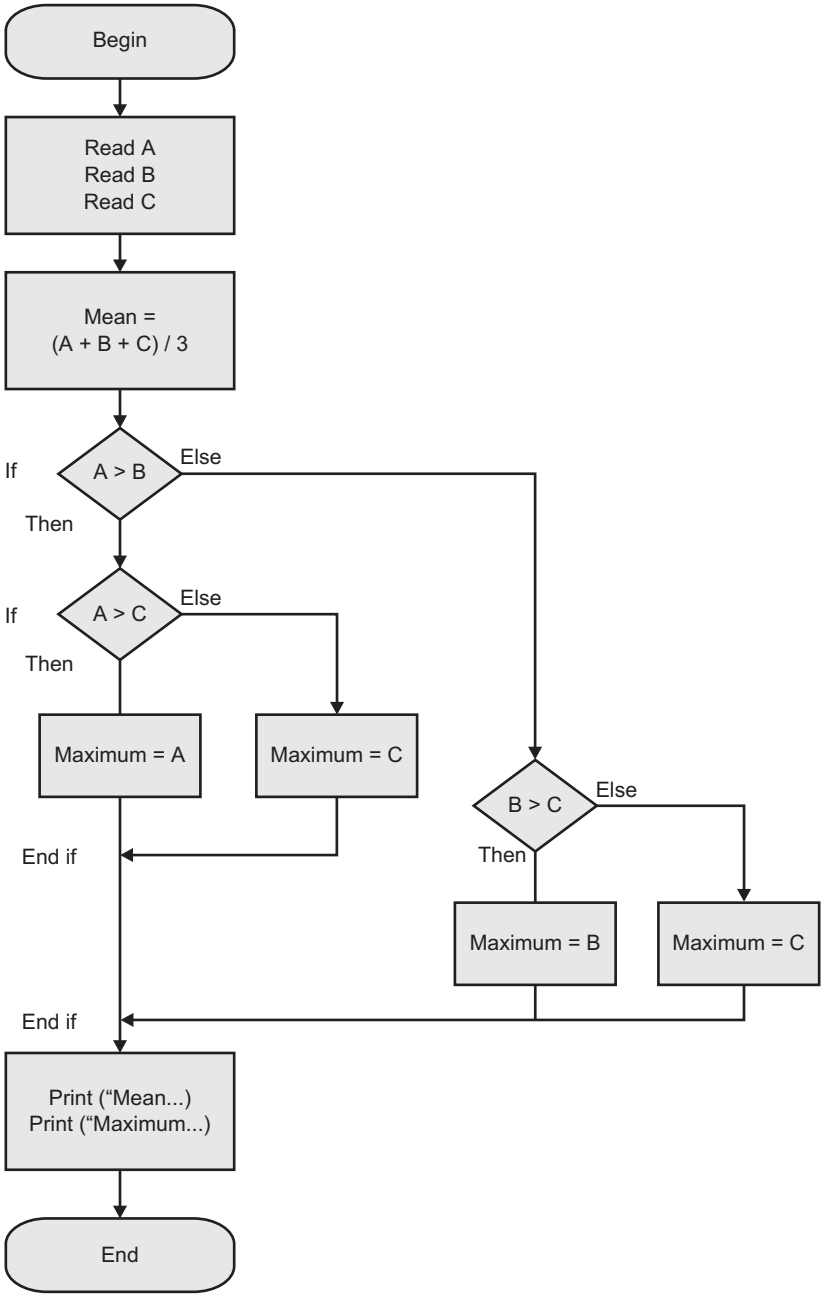
Note one important thing about this code: it has some non-executable statements (those before the **Begin** and those after the **Begin** that are actually blank lines) that we will have to take account of when we come to count the number of executable statements later. The line numbering makes it a little easier to do the counting.

By the way, you may have noticed that the program does not recognise if two of the numbers are the same value, but simplicity is more important than sophistication at this stage.

This program can be expressed as a flow chart; have a go at drawing it before you look at the solution in the text.

Figure 4.7 shows the flow chart for Example 4.5.

Figure 4.7 Flow chart representation for Example 4.5

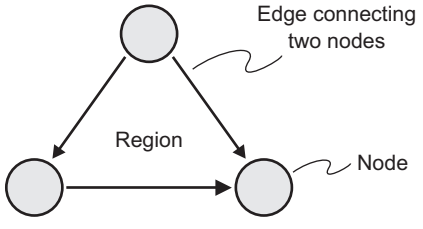


Before we move on to look at how we generate test cases for code, we need to look briefly at another form of graphical representation called the control flow graph.

Control flow graphs

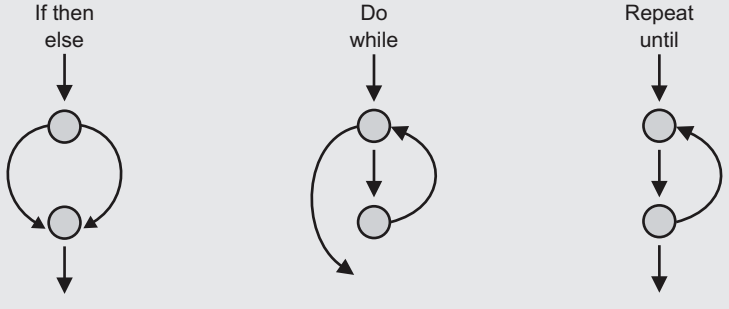
A control flow graph provides a method of representing the decision points and the flow of control within a piece of code, so it is just like a flow chart except that it only shows decisions. A control flow graph is produced by looking only at the statements affecting the flow of control.

The graph itself is made up of two symbols: nodes and edges. A node represents any point where the flow of control can be modified (i.e. decision points), or the points where a control structure returns to the main flow (e.g. END WHILE or ENDIF). An edge is a line connecting any two nodes. The closed area contained within a collection of nodes and edges, as shown in the diagram, is known as a region.



We can draw 'subgraphs' to represent individual structures. For a flow graph the representation of sequence is just a straight line, since there is no decision to cause any branching.

CONTROL FLOW SUBGRAPHS



The subgraphs show what the control flow graph would look like for the program structures we are already familiar with.

Any chunk of code can be represented by using these subgraphs.

DRAWING A CONTROL FLOW GRAPH

The steps are as follows:

1. Analyse the component to identify all control structures; that is, all statements that can modify the flow of control, ignoring all sequential statements.
2. Add a node for any decision statement.
3. Expand the node by substituting the appropriate subgraph representing the structure at the decision point.

As an example, we will return to Example 4.5.

Step 1 breaks the code into statements and identifies the control structures, ignoring the sequential statements, in order to identify the decision points; these are highlighted below.

```
1  Program MaxandMean
2
3       A, B, C, Maximum: Integer
4       Mean: Real
5
6  Begin
7
8       Read A
9       Read B
10      Read C
11      Mean = (A + B + C)/3
12
13          If A > B
14          Then
15          If A > B
16          Then
17          Maximum = A
18          Else
19          Maximum = C
20          Endif
21          Else
22          If B > C
23          Then
24          Maximum = B
25          Else
26          Maximum = C
27          Endif
28          Endif
29
30      Print ("Mean of A, B and C is ", Mean)
31      Print ("Maximum of A, B, C is ", Maximum)
32
33 End
```

Step 2 adds a node for each branching or decision statement (Figure 4.8).

Step 3 expands the nodes by substituting the appropriate subgraphs (Figure 4.9).

Figure 4.8 Control flow graph showing subgraphs as nodes

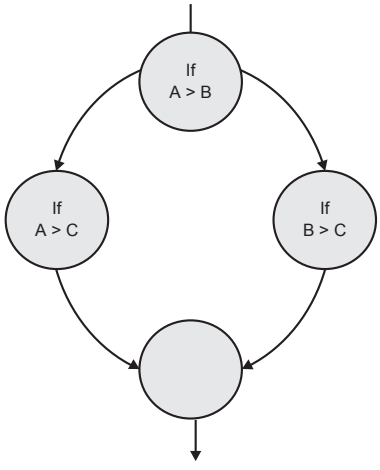
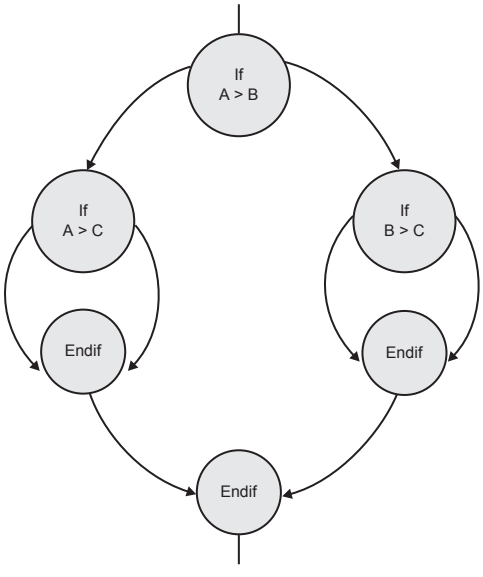


Figure 4.9 Control flow graph with subgraphs expanded



CHECK OF UNDERSTANDING

1. What is the difference between a flow chart and a control flow graph?
2. Name the three fundamental program structures that can be found in programs.
3. Briefly explain what is meant by an edge, a node and a region in a control flow graph.

Exercise 4.6

Draw a flow chart and a control flow graph to represent the following code:

```
1  Program OddandEven
2
3       A, B: Real;
4       Odd: Integer;
5
6  Begin
7       Read A
8       Read B
9       C = A + B
10      D = A – B
11      Odd = 0
12
13      If A/2 DIV 2 <> 0  (DIV gives the remainder after division)
14      Then Odd = Odd + 1
15      Endif
16
17      If B/2 DIV 2 <> 0
18      Then Odd = Odd + 1
19      Endif
20
21          If Odd = 1
22          Then
23          Print ("C is odd")
24          Print ("D is odd")
25          Else
26          Print ("C is even")
27          Print ("D is even")
28          Endif
29
30 End
```

The answer can be found at the end of the chapter.

Statement testing and coverage

Statement testing is testing aimed at exercising programming statements. If we aim to test every executable statement, we call this full or 100 per cent statement coverage. If we exercise half the executable statements this is 50 per cent statement coverage, and

so on. Remember: we are only interested in executable statements, so we do not count non-executable statements at all when we are measuring statement coverage.

Why measure statement coverage? It is a very basic measure that testing has been (relatively) thorough. After all, a suite of tests that had not exercised all of the code would not be considered complete. Actually, achieving 100 per cent statement coverage does not tell us very much, and there are much more rigorous coverage measures that we can apply, but it provides a baseline from which we can move on to more useful coverage measures. Look at the following pseudo code:

```
1  Program Coverage example
2      A, X: Integer
3  Begin
4      Read A
5      Read X
6      If A > 1 AND X = 2
7      Then
8      X = X/A
9      Endif
10     If A = 2 OR X = 2
11     Then
12     X = X + 1
13     Endif
14 End
```

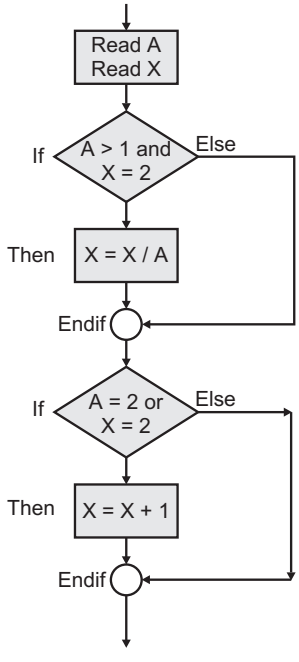
A flow chart can represent this, as in Figure 4.10.

Having explored flow charts and flow graphs a little, you will see that flow charts are very good at showing you where the executable statements are; they are all represented by diamonds or rectangles and where there is no rectangle, there is no executable code. A flow graph is less cluttered, showing only the structural details, in particular where the program branches and rejoins. Do we need both diagrams? Well, neither has everything that we need. However, we can produce a version of the flow graph that allows us to determine statement coverage.

To do this we build a conventional control flow graph but then we add a node for every branch in which there are one or more statements. Take the Coverage example: we can produce its flow graph easily as shown in Figure 4.11.

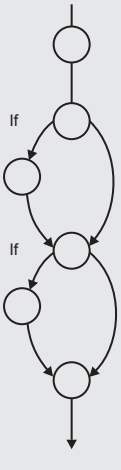
Before we proceed, let us confirm what happens when a program runs. Once the program starts, it will run through to the end executing every statement that it comes to in sequence. Control structures will be the only diversion from this end-to-end sequence, so we need to understand what happens with the control structures when the program runs. The best way to do that is to 'dry run' the program with some inputs; this means writing down the inputs and then stepping through the program logic, noting what happens at each step and what values change. When you get to the end, you will know what the output values (if any) will be and you will know exactly what path the program has taken through the logic.

Figure 4.10 Flow chart for Coverage example



THE HYBRID FLOW GRAPH

Figure 4.11 The hybrid flow graph

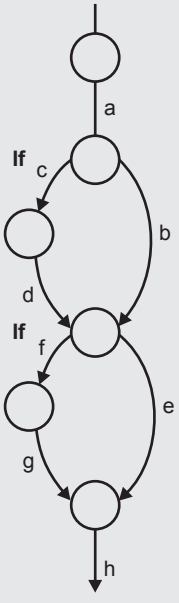


Note the additional nodes that represent the edges with executable statements in them; they make it a little easier to identify what needs to be counted for statement coverage.

PATHS THROUGH A PROGRAM

Flow charts, control flow graphs and hybrid flow graphs all show essentially the same information, but sometimes one format is more helpful than another. We have identified the hybrid flow graph as a useful combination of the control flow graph and the control flow chart. To make it even more useful we can add to it labels to indicate the paths that a program can follow through the code. All we need to do is to label each edge; paths are then made up from sequences of the labels, such as abeh, which make up a path through the code (see Figure 4.12).

Figure 4.12 Paths through the hybrid flow graph example



In the Coverage example, for which we drew the flow chart in Figure 4.10, 100 per cent statement coverage can be achieved by writing a single test case that follows the path acdfgh (using lower case letters to label the arcs on the diagram that represent path fragments). By setting A = 2 and X = 2 at point a, every statement will be executed once. However, what if the first decision should be an OR rather than an AND? The test would not have detected the error, since the condition will be true in both cases. Similarly, if the second decision should have stated X > 2, this error would have gone undetected because the value of A guarantees that the condition is true. Also, there is a path through the program in which X goes unchanged (the path abeh). If this were an error, it would also go undetected.

Remember that statement coverage takes into account only executable statements. There are 12 in the Coverage example if we count the **BEGIN** and **END** statements, so

statement coverage would be 12/12 or 100 per cent. There are alternative ways to count executable statements: some people count the **BEGIN** and **END** statements; some count the lines containing **IF, THEN** and **ELSE**; some count none of these. It does not matter as long as:

- You exclude the non-executable statements that precede **BEGIN**.
- You ignore blank lines that have been inserted for clarity.
- You are consistent about what you do or do not include in the count with respect to control structures.

As a general rule, for the reasons given above, statement coverage is too weak to be considered an adequate measure of test effectiveness.

STATEMENT TESTING – EXAMPLE 4.6

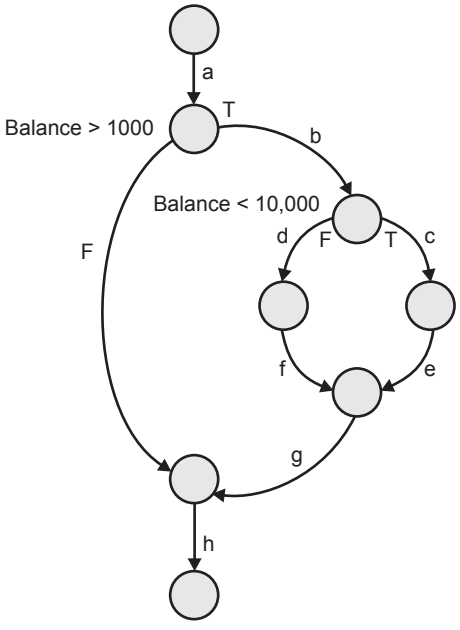
Here is an example of the kind you might see in an exam. Try to answer the question, but if you get stuck the answer follows immediately in the text.

Here is a program. How many test cases will you need to achieve 100 per cent statement coverage and what will the test cases be?

```
1  Program BestInterest
2      Interest, Base Rate, Balance: Real
3
4  Begin
5      Base Rate = 0.035
6      Interest = Base Rate
7
8          Read (Balance)
9          If Balance > 1000
10         Then
11         Interest = Interest + 0.005
12         If Balance < 10000
13         Then
14         Interest = Interest + 0.005
15         Else
16         Interest = Interest + 0.010
17         Endif
18         Endif
19
20 Balance = Balance × (1 + Interest)
21
22 End
```

Figure 4.13 shows what the flow graph looks like. It is drawn in the hybrid flow graph format so that you can see which branches need to be exercised for statement coverage.

Figure 4.13 Paths through the hybrid flow graph – Example 4.6



It is clear from the flow graph that the left-hand side (Balance below £1,000) need not be exercised, but there are two alternative paths (Balance between £1,000 and £10,000 and Balance > £10,000) that need to be exercised.

So, we need two test cases for 100 per cent statement coverage and Balance = £5,000, Balance = £20,000 will be suitable test cases.

Alternatively, we can aim to follow the paths abcegh and abdfgh marked on the flow graph. How many test cases do we need to do that?

We can do this with one test case to set the initial balance value to a value between £1,000 and £10,000 (to follow abcegh) and one test case to set the initial balance to something higher than £10,000, say £12,000 (to follow path abdfgh).

So, we need two test cases to achieve 100 per cent statement coverage in this case.

Now look at this example from the perspective of the tester actually trying to achieve statement coverage. Suppose we have set ourselves an exit criterion of 100 per cent statement coverage by the end of component testing. If we ran a single test with an input of Balance = £10,000 we can see that that test case would take us down the path abdfgh, but it would not take us down the path abcegh, and line 14 of the pseudo code would not be exercised. So that test case has not achieved 100 per cent statement coverage and we will need another test case to exercise line 14 by taking path abcegh. We know that Balance = £5,000 would do that. We can build up a test suite in this way to achieve any desired level of statement coverage.

CHECK OF UNDERSTANDING

1. What is meant by statement coverage?
2. In a flow chart, how do you decide which paths to include in determining how many test cases are needed to achieve a given level of statement coverage?
3. Does 100 per cent statement coverage exercise all the paths through a program?

Exercise 4.7

For the following program:

```
1 Program Grading
2
3       StudentScore: Integer
4       Result: String
5
6 Begin
7
8 Read StudentScore
9
10 If StudentScore > 79
11 Then Result = "Distinction"
12 Else
13      If StudentScore > 59
14      Then Result = "Merit"
15      Else
16          If StudentScore > 39
17          Then Result = "Pass"
18          Else Result = "Fail"
19          Endif
20      Endif
21 Endif
22 Print ("Your result is", Result)
23 End
```

How many test cases are needed for 100 per cent statement coverage?

The answer can be found at the end of the chapter.

Exercise 4.8

Now using the program Grading in Exercise 4.7 again, try to calculate whether 100 per cent statement coverage is achieved with a given set of data.

Suppose we ran two test cases, as follows:

Test Case 1 StudentScore = 50

Test Case 2 StudentScore = 30

1. Would 100 per cent statement coverage be achieved?
2. If not, which lines of pseudo code will not be exercised?

The answer can be found at the end of the chapter.

Decision testing and coverage

Decision testing aims to ensure that the decisions in a program are adequately exercised. Decisions, as you know, are part of selection and iteration structures; we see them in IF THEN ELSE constructs and in DO WHILE or REPEAT UNTIL loops. To test a decision, we need to exercise it when the associated condition is true and when the condition is false; this guarantees that both exits from the decision are exercised.

As with statement testing, decision testing has an associated coverage measure and we normally aim to achieve 100 per cent decision coverage. Decision coverage is measured by counting the number of decision outcomes exercised (each exit from a decision is known as a decision outcome) divided by the total number of decision outcomes in a given program. It is usually expressed as a percentage.

The usual starting point is a control flow graph, from which we can visualise all the possible decisions and their exit paths. Have a look at the following example.

```
1 Program Check
2
3     Count, Sum, Index: Integer
4
5 Begin
6
7     Index = 0
8     Sum = 0
9     Read (Count)
10    Read (New)
11
12        While Index <= Count
13        Do
14        If New < 0
15        Then
16        Sum = Sum + 1
17        Endif
18        Index = Index + 1
19        Read (New)
20        Enddo
21
22    Print ("There were", Sum, "negative numbers in the input stream")
23
24 End
```

This program has a WHILE loop in it. There is a golden rule about WHILE loops. If the condition at the WHILE statement is true when the program reaches it for the first time, then any test case will exercise that decision in both directions because it will eventually be false when the loop terminates. For example, as long as Index is less

than Count when the program reaches the loop for the first time, the condition will be true and the loop will be entered. Each time the program runs through the loop it will increase the value of Index by one, so eventually Index will reach the value of Count and pass it, at which stage the condition is false and the loop will not be entered. So, the decision at the start of the loop is exercised through both its true exit and its false exit by a single test case. This makes the assumption that the logic of the loop is sound, but we are assuming that we are receiving this program from the developers, who will have debugged it.

Now all we have to do is to make sure that we exercise the **If** statement inside the loop through both its true and false exits. We can do this by ensuring that the input stream has both negative and positive numbers in it.

For example, a test case that sets the variable Count to 5 and then inputs the values 1, 5, –2, –3, 6 will exercise all the decisions fully and provide us with 100 per cent decision coverage. Note that this is considered to be a single test case, even though there is more than one value for the variable New, because the values are all input in a single execution of the program. This example does not provide the smallest set of inputs that would achieve 100 per cent decision coverage, but it does provide a valid example.

Although programs with loops are a little more complicated to understand than programs without loops, they can be easier to test once you get the hang of them.

DECISION TESTING – EXAMPLE 4.7

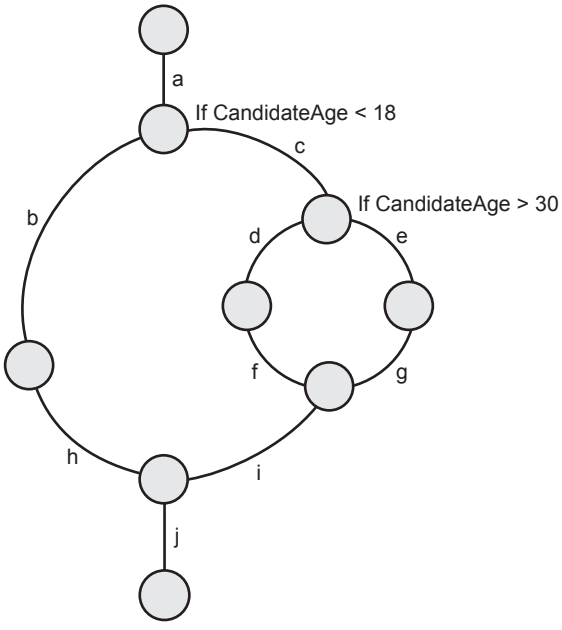
Let us try an example without a loop now.

```
1 Program Age Check
2
3     CandidateAge: Integer;
4
5 Begin
6
7     Read(CandidateAge)
8
9     If CandidateAge < 18
10    Then
11    Print ("Candidate is too young")
12    Else
13    If CandidateAge > 30
14    Then
15    Print ("Candidate is too old")
16    Else
17    Print("Candidate may join Club 18–30")
18    Endif
19    Endif
20
21 End
```

Have a go at calculating how many test cases are needed for 100 per cent decision coverage and see if you can identify suitable test cases.

Figure 4.14 shows the flow graph drawn in the hybrid flow graph format.

Figure 4.14 Paths through the hybrid flow graph – Example 4.7



How many test cases will we need to achieve 100 per cent decision coverage? Well each test case will just run through from top to bottom, so we can only exercise one branch of the structure at a time.

We have labelled the path fragments a, b, c, d, e, f, g, h, i, j and you can see that we have three alternative routes through the program – path abhj, path acegij and path acdfij. That needs three test cases.

The first test case needs decision 1 to be true – so CandidateAge = 16 will be OK here. The second needs to make the first decision false and the second decision true, so CandidateAge must be more than 18 and more than 30 – let us say 40. The third test case needs the first decision to be false and the second decision to be false, so CandidateAge of 21 would do here. (You cannot tell which exit is true and which is false in the second decision; if you want to, you can label the exits T and F, though in this case it does not really matter because we intend to exercise them both anyway.)

So, we need three test cases for 100 per cent decision coverage:

CandidateAge = 16

CandidateAge = 21

CandidateAge = 40

which will exercise all the decisions.

Note that when we are testing decisions, we need to exercise the true and false outcomes of each decision, even if one of these has no statements associated with it. So, decision coverage gives us that little bit extra in return for a little more work. One hundred per cent decision coverage guarantees 100 per cent statement coverage, but 100 per cent statement coverage may be less than 100 per cent decision coverage.

CHECK OF UNDERSTANDING

1. What is the purpose of decision testing?
2. How many test cases are needed to exercise a single decision?
3. How many test cases are needed to exercise a loop structure?

Exercise 4.9

This program reads a list of non-negative numbers terminated by –1.

```
1 Program Counting Numbers
2
3     A: Integer
4     Count: Integer
5
6 Begin
7     Count = 0
8     Read A
9     While A <> -1
10    Do
11    Count = Count + 1
12    Read A
13    EndDo
14
15    Print ("There are", Count, "numbers in the list")
16 End
```

How many test cases are needed to achieve 100 per cent decision coverage?

The answer can be found at the end of the chapter.

Exercise 4.10

Using Program Counting Numbers from Exercise 4.9, what level of decision coverage is achieved by the single input A = –1?

The answer can be found at the end of the chapter.

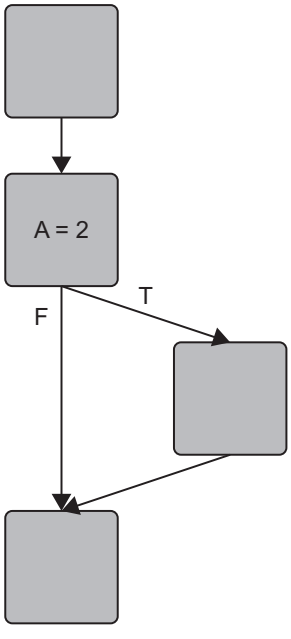
Simplified control flow graphs

Simplified control flow graphs (CFGs) share the format used for flow charts. They incorporate less detail, since their purpose is to identify control structures but not the detail of code. The testing literature sometimes provides examples and questions using simplified CFGs instead of code, so this section deals with the interpretation of simplified CFGs and provides examples and exercises that cover these structures.

In simplified CFGs, a sequence of code is presented as one or more boxes in a line. Strictly speaking, only a single box is needed but additional boxes are sometimes included to make the CFG easier to follow. Decisions are represented by a single box with two exits, usually labelled T and F (for True and False respectively).

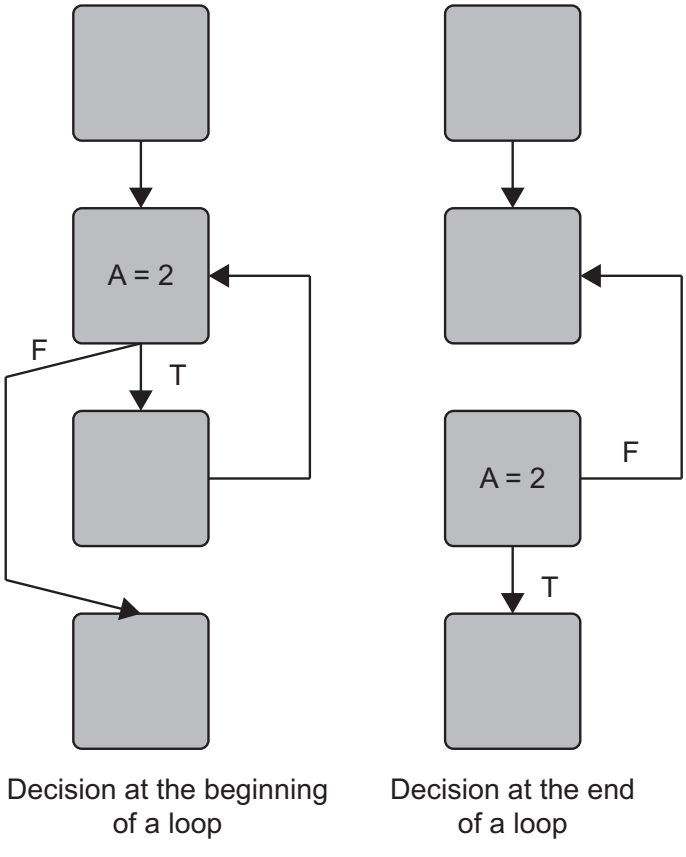
Figure 4.15 shows a simplified CFG depicting a sequence of code followed by a decision based on the evaluation of a decision.

Figure 4.15 Simplified control flow graph: a decision



Looping structures can have a decision at the beginning of the loop or at the end (see Figure 4.16).

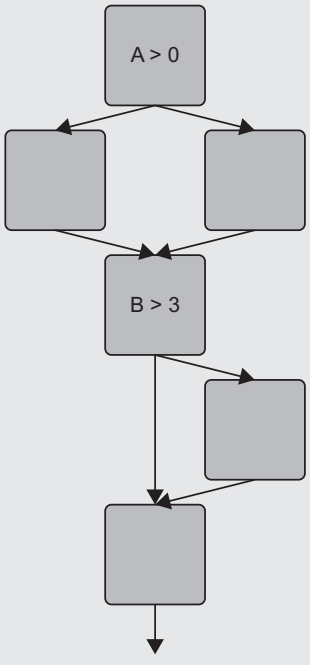
Figure 4.16 Simplified control flow graph: location of a decision in a loop



Here is an example of how simplified CFGs are read and interpreted for simple structures.

SIMPLIFIED CONTROL FLOW GRAPHS – EXAMPLE 4.8

Figure 4.17 Example of how simplified control flow graphs are read and interpreted



Which of the following pairs of tests achieve 100 per cent decision coverage for the structure shown in Figure 4.17?

a. A = 1, B = 2; A = 0, B = 1
b. A = 2, B = 4; A = 1, B = 2
c. A = 3, B = 0; A = 0, B = 1
d. A = 0, B = 4; A = 1, B = 3

Answer – The correct answer is d.

Option a. In the first decision A is greater than zero in the first test and equals zero in the second, so this decision has been exercised in both directions. B is less than 3 in both tests, so decision B > 3 is not exercised in both directions. Therefore 100 per cent decision coverage has not been achieved.

Option b. A is greater than zero in both tests, so the decision A > 0 has not been exercised in both directions. Therefore 100 per cent decision coverage has not been achieved.

Option c. A is greater than zero in test 1 and equals zero in test 2, so decision A > 0 has been exercised in both directions. B is less than 3 in both tests, so decision B > 3 has not been exercised in both directions. Therefore 100 % decision coverage has not been achieved.

Option d. A equals zero on test 1 and equals 1 in test 2, so test A > 0 has been exercised in both directions. B is greater than 3 in test 1, but not in test 2, so decision B > 3 has been exercised in both directions. Therefore 100 per cent decision coverage has been achieved.

SIMPLIFIED CONTROL FLOW GRAPHS – 50 PER CENT DECISION COVERAGE – EXAMPLE 4.9

Using the same diagram as in Example 4.8, which of the following sets of test cases achieve 50 per cent decision coverage?

a. A = 1, B = 2; A = 0, B = 3
b. A = 0, B = 4; A = 1, B = 3
c. A = 1, B = 2; A = 2, B = 2
d. A = 5, B = 4; A = 0, B = 3

Answer – The correct answer is c.

Option a. The decision A > 0 has been exercised in both directions, but decision B > 3 has been exercised in only one direction. Therefore, the decision coverage would be 3/4 or 75 per cent.

Option b. The decision A > 0 has been exercised in both directions and decision B > 3 has been exercised in both directions. Therefore, the decision coverage would be 4/4 or 100 per cent.

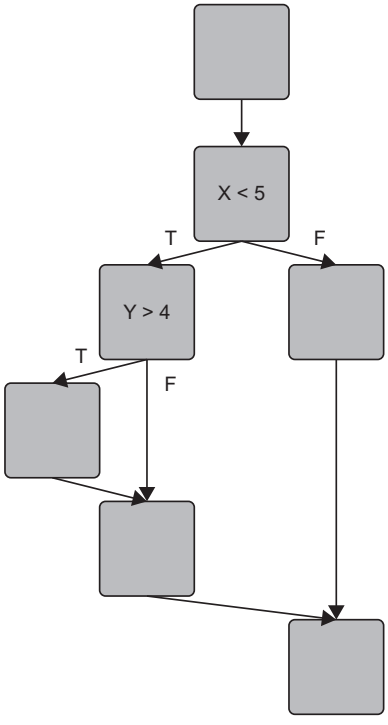
Option c. The decision A > 0 has not been exercised in both directions and decision B > 3 has not been exercised in both directions. Therefore, the decision coverage is 2/4 or 50 per cent.

Option d. The decision A > 0 has been exercised in both directions and decision B > 3 has been exercised in both directions. Therefore, the decision coverage would be 4/4 or 100 per cent.

Exercise 4.11

In the simplified CFG shown in Figure 4.18, which of the following sets of test cases achieve 100 per cent statement coverage?

Figure 4.18 Control flow graph for Exercise 4.11



a. X = 7, Y = 6; X = 3, Y = 5
b. X = 3, Y = 3; X = 4, Y = 4
c. X = 6, Y = 5; X = 1, Y = 1
d. X = 5, Y = 4; X = 4, Y = 4

The answer can be found at the end of the chapter.

Exercise 4.12

In the same CFG shown in the previous exercise, how many test cases are required to achieve 100 per cent decision coverage?

a. 1
b. 2
c. 3
d. 4

The answer can be found at the end of the chapter.

There are further questions for you to try at the end of this chapter.

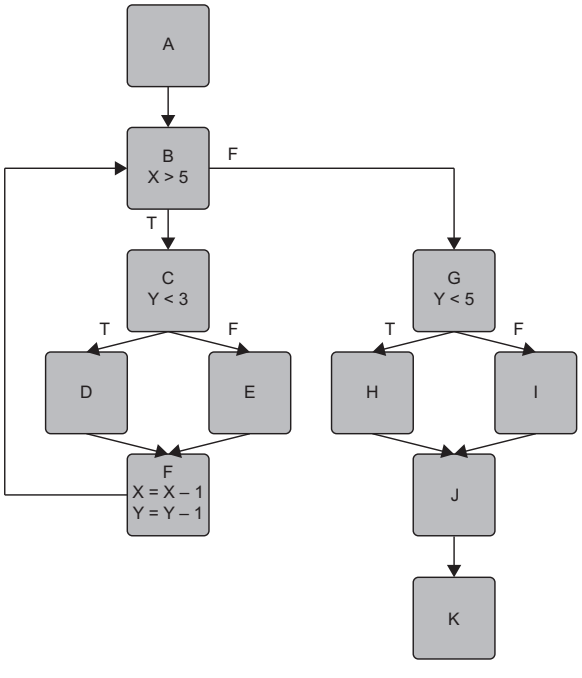
Other white-box test techniques

More sophisticated techniques are available to provide increasingly complete code coverage. In some applications these are essential: for example, in a safety-critical system it is vital to know that nothing unacceptable happens at any point when the code is executed. Would you like to 'fly by wire' if you did not know what was happening in the software? The many well-documented mishaps in computer-controlled systems provide compelling examples of what can happen if code – even code that is not providing essential functionality in some cases – does something unexpected. Measures such as condition coverage and multiple condition coverage are used to reduce the likelihood that code will behave in unpredictable ways by examining more of it in more complex scenarios.

Coverage is also applicable to other types and levels of structure. For example, at the integration level it is useful to know what percentage of modules or interfaces has been exercised by a test suite, while at the functional level it is helpful to step through all the possible paths of a menu structure. We can also apply the idea of coverage to areas outside the computer; for example, by exercising all the possible paths through a business process as testing scenarios.

Exercise 4.13

Figure 4.19 Test case for Exercise 4.13



Using Figure 4.19, test case 1 is executed with the values X = 6, Y = 2. **Which of the following paths will be executed by test case 1?**

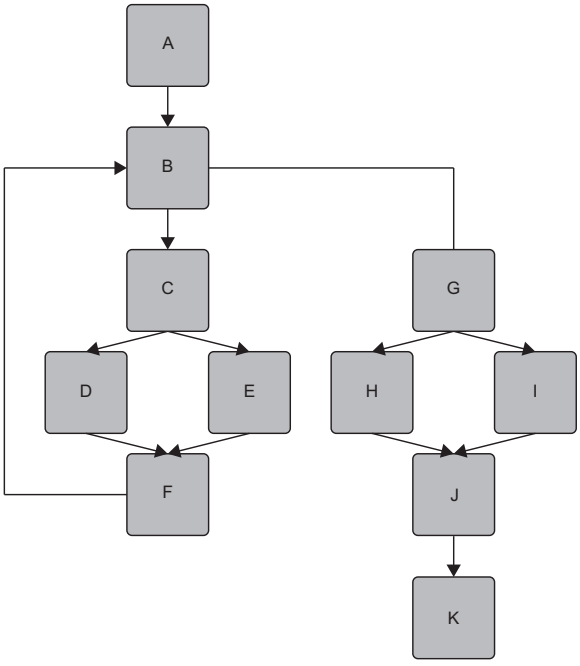
a. ABCEFBGIJK
b. ABCDFBGIJK
c. ABCDFBGHJK
d. ABCDFBCEFGHJK

Exercise 4.14

In Figure 4.20, the goal for the project is 100 per cent decision coverage and the following tests have so far been executed:

- Test 1 covers path ABCDFBGHJK.
- Test 2 covers path ABGIJK.

Figure 4.20 Test case for Exercise 4.14



Which of the following statements about decision coverage is correct?

a. Decision B has not been tested completely.
b. 100 per cent decision coverage has been achieved.
c. Decision G has not been tested completely.
d. Decision C has not been tested completely.

The main examinable material continues from here.

EXPERIENCE-BASED TECHNIQUES

Experience-based techniques are those that you fall back on when there is no adequate specification from which to derive specification-based test cases or no time to run the full structured set of tests. They use the users' and the testers' experience to determine the most important areas of a system and to exercise these areas in ways that are both consistent with expected use (and abuse) and likely to be the sites of errors – this is where the experience comes in. Even when specifications are available, it is worth supplementing the structured tests with some that you know by experience have found defects in other similar systems.

Techniques range from the simplistic approach of ad hoc testing or error guessing through to the more sophisticated techniques such as exploratory testing, but all tap the knowledge and experience of the tester rather than systematically exploring a system against a written specification.

Error guessing

Error guessing is a very simple technique that takes advantage of a tester's skill, intuition and experience with similar applications to identify special tests that may not be easy to capture by the more formal techniques. When applied after systematic techniques, error guessing can add value in identifying and exercising test cases that target known or suspected weaknesses or that simply address aspects of the application that have been found to be problematical in the past.

The main drawback of error guessing is its varying effectiveness, depending as it does on the experience of the tester deploying it. However, if several testers and/or users contribute to constructing a list of possible errors and tests are designed to attack each error listed, this weakness can be effectively overcome. Another way to make error guessing more structured is by the creation of defect and failure lists. These lists can use available defect and failure data (where this exists) as a starting point, but the list can be expanded by using the testers' and users' experience of why the application under test in particular is likely to fail. The defect and failure list can be used as the basis of a set of tests that are applied after the systematic techniques have been used. This systematic approach is known as fault attack.

Exploratory testing

Exploratory testing is a technique that combines the experience of testers with a structured approach to testing where specifications are either missing or inadequate and where there is severe time pressure. It exploits concurrent test design, test execution, test logging and learning within time-boxes and is structured around a test charter containing test objectives. In this way exploratory testing maximises the amount of testing that can be achieved within a limited time frame, using test objectives to maintain focus on the most important areas.

Checklist-based testing

Checklist-based testing is testing based on high-level checklists, which may be drawn from many sources and generally encapsulate individuals' experience and information gleaned from other sources, such as standards, known problem areas, expected use scenarios and any other relevant sources. Checklists are a guide to the testing required, so are used mainly by experienced testers as a source of ideas from which detailed tests are derived.

The checklist is drawn up during test analysis as a set of test conditions and may reuse or update previous checklists. Checklists can support both functional and non-functional testing. Coverage is defined by completion of the checklist and may be higher than for other forms of testing, though repeatability will typically be lower than for more formal testing.

SYSTEMATIC AND EXPERIENCE-BASED TECHNIQUES

How do we decide which is the best technique? There are some simple rules of thumb:

1. Always make functional testing the first priority. It may be necessary to test early code products using structural techniques, but we only really learn about the quality of software when we can see what it does.
2. When basic functional testing is complete that is a good time to think about test coverage. Have you exercised all the functions, all the requirements, all the code? Coverage measures defined at the beginning as exit criteria can now come into play. Where coverage is inadequate, extra tests will be needed.
3. Use structural methods to supplement functional methods where possible. Even if functional coverage is adequate, it will usually be worth checking statement and decision coverage to ensure that enough of the code has been exercised during testing.
4. Once systematic testing is complete, there is an opportunity to use experience-based techniques to ensure that all the most important and most error-prone areas of the software have been exercised. In some circumstances, such as poor specifications or time pressure, experience-based testing may be the only viable option.

CHECK OF UNDERSTANDING

1. What is meant by experience-based testing?
2. Briefly compare error guessing and exploratory testing.
3. When is the best time to use experience-based testing?

SUMMARY

In this chapter we have considered the most important terminology needed in discussing the specification stage of a generalised test process, which was introduced in **Chapter 1**. We explained how test conditions are derived and how test cases can be designed and grouped into test procedures for execution.

Test design techniques were categorised into three main groups known as specification-based or black-box techniques, structure-based or white-box techniques, and experience-based techniques.

The specification-based techniques introduced were equivalence partitioning, boundary value analysis, state transition testing, decision table testing and use case testing. Specific worked examples of all except use case testing were given (and this was excluded solely because the examination does not require the ability to generate test cases from use cases). Structure-based techniques were introduced and worked examples were given for statement testing and decision testing. Experience-based techniques introduced included error guessing and exploratory testing.

Finally, the factors involved in selecting test case design techniques were discussed and guidance given on the selection criteria to be applied.

Example examination questions with answers

E1. K2 question
Which of the following correctly characterises white-box test techniques?

a. Test cases may be used to detect differences between requirements and implementation.

b. Test cases may be used to determine deviations from requirements.

c. Test cases may be based on software architecture and used to exercise interfaces.

d. Test cases may be based on user stories and used to exercise use cases.

E2. K2 question
Which of the following identifies a key difference between black-box and white-box test techniques?

a. Coverage measures are applied to white-box test cases but not to black-box test cases.

b. Black-box test cases may be based on requirements or on the tester's experience; white-box test cases are based on structure.

c. Coverage measures are applied to black-box test cases but not to white-box test cases.

d. Black-box tests can determine deviations from requirements; white-box test can determine deviations from design.

E3. K3 question

A washing machine has three temperature bands for different kinds of fabrics: fragile fabrics are washed at temperatures between 15 and 30 degrees Celsius; normal fabrics are washed at temperatures between 31 and 60 degrees Celsius; heavily soiled and tough fabrics are washed at temperatures between 61 and 100 degrees Celsius.

Which of the following contains only values that are in *different* equivalence partitions?

a. 15, 30, 60

b. 20, 35, 60

c. 25, 45, 75

d. 12, 35, 55

E4. K2 question
Which of the following correctly identifies the derivation of test cases in use case testing?

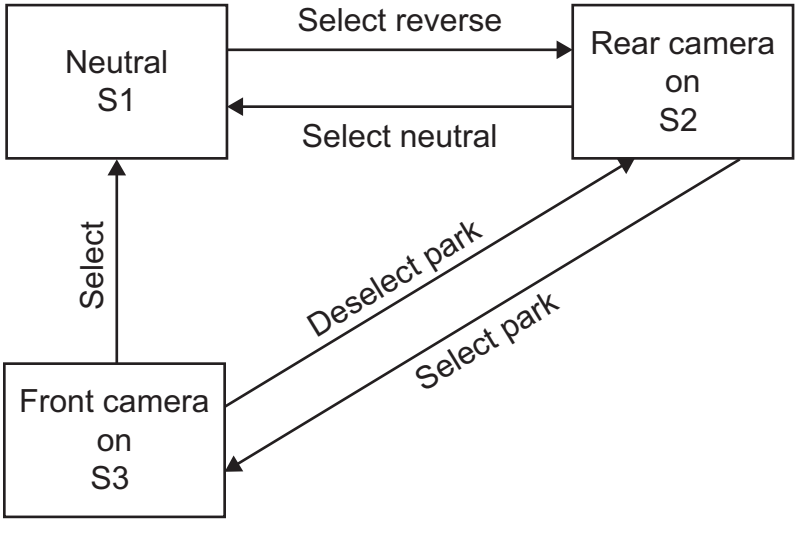
a. Test cases are derived solely from interactions between human actors and system subjects.

b. Test cases can be based on interactions and activities represented by workflows, which must be represented graphically.

c. Test cases can be derived from defined behaviours, alternative or exceptional behaviours, or error handling behaviours.

d. Test cases specify behaviour that a single subject can perform in collaboration with a single actor.

E5. K3 question
Which *one* of the following statements is correct?



Test case	1	2	3	4	5	6
Start state	S1	S1	S2	S2	S3	S3
Input	Select reverse	Select park	Select park	Select neutral	Select neutral	Deselect park
Expected final state	S2	S3	S3	S1	S1	S2

a. The given test cases cover both valid and invalid transitions in the state transition diagram.

b. The given test cases represent only the possible valid transitions in the state transition diagram.

c. The given test cases represent only some of the valid transitions in the state transition diagram.

d. The given test cases represent sequential pairs of transitions in the state transition diagram.

Answers to questions in the chapter

SA1. The correct answer is d.
SA2. The correct answer is d.
SA3. The correct answer is b.

Exercise 4.1

The partitions are: £0.00–£1,000.00, £1,000.01–£2,000.00, and >= £2,000.01.

Exercise 4.2

The valid partitions are: £0.00–£20.00, £20.01–£40.00, and >= £40.01. Non-valid partitions would include negative values and alphabetic characters.

Exercise 4.3

The partitions are: question scores 0–20; total 0–100; question differences: 0–3 and > 3; total differences 0–10 and > 10.

Boundary values are: –1, 0, 1 and 19, 20, 21 for the question scores; –1, 0, 1 (again) and 99, 100, 101 for the question paper totals; –1, 0, 1 (again) and 2, 3, 4 for differences between question scores for different markers; and –1, 0, 1 (again) and 9, 10, 11 for total differences between different markers.

In this case, although the –1, 0, 1 values occur several times, they may be applied to different parts of the program (e.g. the question score checks will probably be in a different part of the program from the total score checks) so we may need to repeat these values in the boundary tests.

Exercise 4.4

Billy will be eligible for a cash payment but not for a share allocation.

Exercise 4.5

The correct answer is b.

Option a includes the transition DE; option c includes the transition CE; option d includes the transition FA. None of these is valid from the diagram.

Exercise 4.6

The flow chart is shown in Figure 4.21. The control flow graph is shown in Figure 4.22.

Exercise 4.7

The answer is four because there are three decisions and every outcome has an executable statement in it.

Figure 4.21 Flow chart for Exercise 4.6

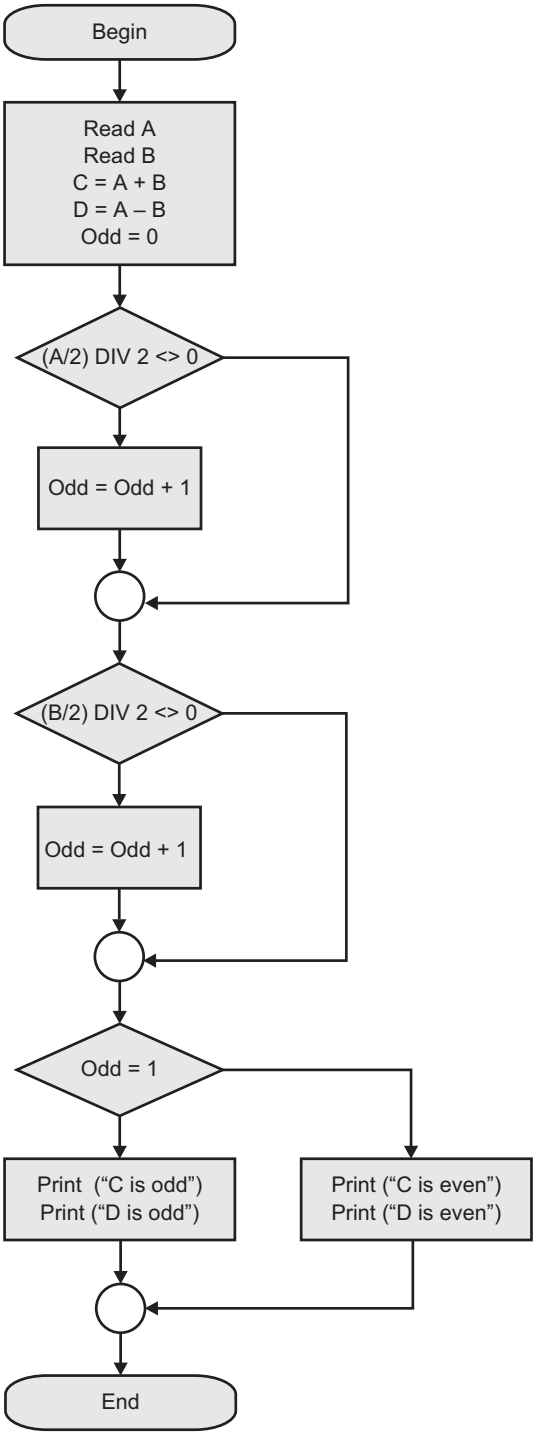
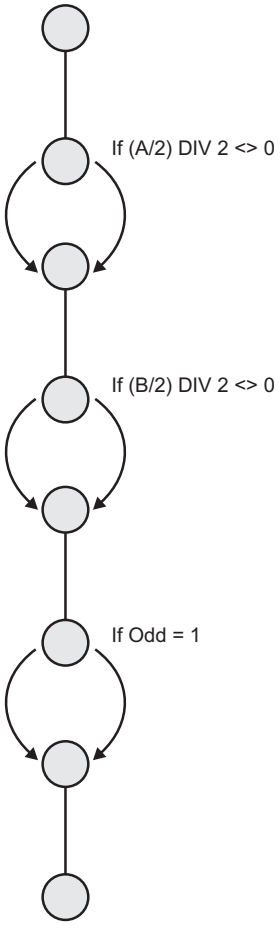


Figure 4.22 Control flow graph for Exercise 4.6



Exercise 4.8

1. No, 100 per cent statement coverage would not be achieved. We know from Exercise 4.7 that four test cases are needed.
2. Statements 11 and 14 would not be exercised because they need inputs higher than 79 and 59 respectively.

Exercise 4.9

The answer is one because a single list terminated by –1 (say 4, 6, 3, –1) will enter the loop the first three times and then exit on the fourth; hence the WHILE decision will be true three times and then false, which exercises the decision in both directions with one test case.

A single test case with values of 1, –1 would also exercise all decisions.

Exercise 4.10

Decision coverage of 50 per cent will be achieved. The –1 input will make the **While** condition False and the loop will not be entered. The program will print the message 'There are 0 integers in the list' and terminate, so the True outcome of the decision will not be exercised.

Exercise 4.11

The correct answer is a.

There are statements in the false exit from decision X < 5 and in the true exit from decision Y > 4, so we need two tests because the test that makes X < 5 false will bypass the decision Y > 4. One of the test cases can be any pair of values that has X >= 5, so X can be 5 or more and Y is irrelevant. In the other test, X must be less than 5 and Y must also be greater than 4. Many pairs of values, such as X = 4, Y = 5, would be suitable. The two tests must have X >= 5 in one test and X < 5 AND Y > 4 in the other. Option a meets these criteria. Option b has X < 5 in both tests, so it cannot be correct. Option c has one test with X > 5 but, although the other test has X < 5, it also sets Y to 1, so Y > 4 is not true. Option d has X = 5 in one test and X < 5 in the other, but in this case the value of Y is 4, so Y > 4 is not true.

Exercise 4.12

The correct answer is c.

In this case we must not only execute every branch with a statement in it, but we must exercise every decision in both its true and false states. We could arrive at the correct answer by recognising that the CFG has two regions and two decisions. The questions regions + 1 and decisions + 1 both give the answer 3. We could also work it out from first principles. The paths we would need to follow would be:

1. False through X < 5
2. True through X < 5 AND True through Y > 4
3. True through X < 5 AND False through Y > 4

These three tests would exercise both decisions in their true and false states and would therefore achieve 100 per cent decision coverage.

Exercise 4.13

The correct answer is c.

For a to be true, Y >= 3 would have to be true. For b to be true, Y >= 5 would have to be true. For d to be true, Y >= 3 would have to be true after one traversal of the loop.

Exercise 4.14

The correct answer is d.

Option a is not true because B was exercised one way in test case 1 and the other way in test case 2. Option b is not true because E has not been traversed, which means that decision E has not been exercised in both directions. Option c is not true because decision G is exercised one way in test case 1 and the other way in test case 2.

Answers to example examination questions

E1. The correct answer is c.
White-box techniques are about the structure of software solutions, so they typically exercise interfaces and check that the software architecture has been correctly implemented. They do not reference requirements as such, and therefore cannot determine deviations from requirements or whether requirements have been correctly implemented. User stories are a form of requirements, so white-box testing is not appropriate to user stories or use cases.

E2. The correct answer is d.
Answer a is incorrect because coverage measures can be applied to both black-box and white-box tests. Answer b is partially correct, in that white-box test cases are based on structure, and black-box test cases are based on requirements but not the tester's experience. Experience-based testing is based on the tester's experience. Option c is incorrect for the same reason as option a is incorrect. Option d is the correct answer because black-box testing is based on requirements and white-box testing is based on design.

E3. The correct answer is c.
Option a includes two values from the lower partition; option b contains two values from the second partition; option d contains one value that is invalid (out of range).

E4. The correct answer is c.
Option a is incorrect because test cases are not solely based on interactions and may also incorporate preconditions and postconditions. Option b is incorrect because workflows are not the only option for describing interactions; activity diagrams or business process models are other alternatives. Option d is incorrect because a use case may represent interactions between a subject and one or more actors. Option c is correct; use cases may represent defined behaviours, alternative or exceptional behaviours, and error handling behaviours.

E5. The correct answer is a.
Option a is correct because the test cases represent all the valid transitions and one possible invalid transition. Option b is incorrect because the test cases recognise one invalid transition. Option c is incorrect because the test cases represent all of the valid transitions. Option d is incorrect because none of the test cases represents sequential pairs of transitions.

5 TEST MANAGEMENT

Geoff Thompson

INTRODUCTION

This chapter provides a generic overview of how testing is organised and how testing is managed within organisations. A generic view of testing will, inevitably, not match the way testing is organised in specific organisations. The issues addressed are nevertheless important for any organisation and need to be considered by all.

We will start by looking at how testing and risk fit together, as well as providing detailed coverage of test planning and the control of testing, and we will identify how independence assists the test process. One very important area in managing the test process is the understanding of the different roles and tasks associated with the testing role such as the test manager and the tester.

We cannot, in one chapter, provide all the knowledge required to enable the reader to become a practising test manager, but we do aim to provide the background information necessary for a reader to understand the various facets of the test management role.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions that follow the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

Test organization

- FL-5.1.1 Explain the benefits and drawbacks of independent testing.
- FL-5.1.2 Identify the tasks of a test manager and tester. (K1)

Test planning and estimation

- FL-5.2.1 Summarize the purpose and content of a test plan. (K2)
- FL-5.2.2 Differentiate between various test strategies. (K2)

- FL-5.2.3 Give examples of potential entry and exit criteria. (K2)
- FL-5.2.4 Apply knowledge of prioritization, and technical and logical dependencies, to schedule test execution for a given set of test cases.
- FL-5.2.5 Identify factors that influence the effort related to testing. (K1)
- FL-5.2.6 Explain the difference between two estimation techniques: the metrics-based technique and the expert-based technique. (K2)

Test monitoring and control

- FL-5.3.1 Recall metrics used for testing. (K1)
- FL-5.3.2 Summarize the purposes, contents, and audiences for test reports.

Configuration management

- FL-5.4.1 Summarize how configuration management supports testing.

Risks and testing

- FL-5.5.1 Define risk level by using likelihood and impact. (K1)
- FL-5.5.2 Distinguish between project and product risks.
- FL-5.5.3 Describe, by using examples, how product risk analysis may influence the thoroughness and scope of testing.

Defect management

- FL-5.6.1 Write a defect report, covering defects found during testing.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K1)
Which of the following is a valid exit criterion from the test execution phase?

a. All tests have been defined.
b. All defects reported have been corrected.
c. All planned tests have been executed.
d. All testing tasks have been assigned.

Question SA2 (K2)
Which of the following are *most* likely to be used when developing a test strategy or test approach?

i. Failure-based approach.
ii. Test specification approach.
iii. Model-based approach.
iv. Analytical-based approach.

a. iii and iv.
b. i and iv.
c. ii and i.
d. i and iii.

Question SA3 (K1)
What can a risk-based approach to testing provide?

a. The types of test techniques to be employed.
b. The total tests needed to provide 100 per cent coverage.
c. An estimation of the total cost of testing.
d. Only that test execution is effective at reducing risk.

RISK AND TESTING

It is not possible to talk about test management without first looking at risk and how it affects a generic test process as defined in **Chapter 1**. If there were no risk of adverse future events in software or hardware development, then there would be no need for testing. In other words, if risks did not exist then neither would testing.

Risk can be defined as the chance of an event, hazard, threat or situation occurring and its undesirable consequences:

> Risk – a factor that could result in future negative consequences, usually expressed as impact and likelihood.

In a project, a test manager will manage two different types of risk: project and product. In both instances the calculation of the risk will be:

> Level of risk = probability of the risk occurring × impact if it did happen

Project risks

While managing the testing project, a test manager will use project risks to manage the capability to deliver.

Project risks include:

- Project issues:
  - Delays in delivery.
  - Inaccurate estimates.
  - Late changes.
- Organisational issues:
  - Skills and training or staff may be inadequate.
  - Personal issues between staff impacting progress.
  - Users, business staff or subject matter experts may be unavailable when needed.
- Political issues:
  - Testers may not communicate their needs and/or test results adequately.
  - Developers and/or testers may fail to follow up on information found in testing and reviews; for example not following up on process improvements identified.
  - There may be an improper attitude to or understanding of the value of testing.
- Supplier issues:
  - Failure of a third party to deliver on time or at all.
  - Contractual issues, such as meeting acceptance criteria.
- Technical issues:
  - Problems in defining the right requirements.
  - The extent that requirements can be met given existing project constraints.
  - Test environment not ready on time.
  - Late data conversion, migration planning and development, and testing data conversion/migration tools.
  - Weakness in the development process that impacts the quality of the work products.
  - Poor defect management resulting in an increase in technical debt.
  - Low quality of the design, code, configuration data, test data and tests.

For each risk found, a probability (chance of the risk being realised) and impact (what will happen if the risk is realised) should be identified as well as the identification and

management of any mitigating actions (actions aimed at reducing the probability of a risk occurring, or reducing the impact of the risk if it did occur).

So, for example, if there was a risk identified that the third-party supplier may be made bankrupt during the development, the test manager would review the supplier's accounts and might decide that the probability of this is medium (1 on a scale of 1 to 5, 1 being a high risk and 5 a low one). The impact on the project if this did happen would be very high (1 using the same scale). The level of risk is therefore 3 × 1 = 3. The lower the number, the more the risk. With 3 being in the medium risk area, the test manager would now have to consider what mitigating actions to take to try to stop the risk becoming a reality. This might include not using the third party or ensuring that payment for third-party deliverables is made efficiently.

When analysing, managing and mitigating these risks, the test manager is following well-established project management principles provided within project management methods and approaches. The project risks recognised during test planning should be documented in the test plan (see later in this chapter for details of the test plan); for the ongoing management and control of existing and new project risks, a risk register should be maintained by the test manager.

Product risks

When planning and defining tests, a test manager or tester using a risk-based testing approach will be managing product risks.

Product risks are risks to the quality of the product. In other words, the potential of a defect occurring in the live environment is a product risk. Examples of product risks are:

- Failure-prone software delivered – not able to perform as intended according to the specification and/or the user requirements.
- System architecture may not adequately support non-functional requirement(s) (e.g. security, reliability, usability, performance).
- A particular computation may be performed incorrectly in certain circumstances.
- A loop structure may be coded incorrectly.
- Feedback from users indicates that the product may not meet expectations.
- The potential that a defect in the software/hardware could cause harm to an individual or company.
- Poor data integrity and quality (e.g. data migration issues, data conversion problems, data transport problems, violation of data standards).
- Software that does not perform its intended functions.

Risks are used to decide where to start testing in the Software Development Life Cycle; for example, the risk of poor requirements could be mitigated by the use of formal reviews as soon as the requirements have been documented at the start of a project. Product risks also provide information enabling decisions regarding how much testing should be carried out on specific components or systems; for example, the more risk

there is, the more detailed and comprehensive the testing may be. In these ways testing is used to reduce the risk of an adverse effect (defect) occurring or being missed.

Mitigating product risks may also involve non-test activities. For example, in the poor requirements situation, a better and more efficient solution may be simply to replace the analyst who is writing the poor requirements in the first place.

As already stated, a risk-based approach to testing provides proactive opportunities to reduce the levels of product risk starting in the initial stages of a project. It involves the identification of product risks and how they are used to guide the test planning, specification and execution. In a risk-based approach, the risks identified:

- will determine the test techniques to be employed, and/or the extent of testing to be carried out; for example, the Motor Industry Software Reliability Association (MISRA) defines which test techniques should be used for each level of risk: the higher the risk, the higher the coverage required from test techniques;
- will determine the levels and types of testing to be performed, such as security testing or accessibility testing;
- will determine the extent of testing to be carried out; for example what the depth of test coverage should be;
- prioritise testing in an attempt to find the critical defects as early as possible; for example, by identifying the areas most likely to have defects (the most complex) the testing can be focused on these areas;
- will determine any non-test activities that could be employed to reduce risk; for example, to provide training to inexperienced designers.

Risk-based testing draws on the collective knowledge and insights of the project stakeholders, testers, designers, technical architects, business reps and anyone with knowledge of the solution to determine the risks and the levels of testing required to address those risks.

To ensure that the chance of a product failure is minimised, risk management activities provide a disciplined approach:

- To analyse (and re-evaluate on a regular basis) what can go wrong. Reviews of existing product risks and looking for any new product risks should occur periodically throughout the life cycle.
- To determine what risks are important to deal with (probability × impact). As the project progresses, owing to the mitigation activities, risks may reduce in importance, or disappear altogether.
- To implement actions to deal with those risks (mitigating actions).
- To make contingency plans to deal with risks should they become actual events.

Testing supports the identification of new risks by continually reviewing risks of the project deliverables throughout the life cycle; it may also help to determine what risks are important to reduce by setting priorities; it may lower uncertainty about risks by,

for example, testing a component and verifying that it does not contain any defects; and lastly by running specific tests it may verify other strategies that deal with risks, such as contingency plans.

Testing is a risk control activity that provides feedback about the residual risk in the product by measuring the effectiveness of critical defect removal and by reviewing the effectiveness of contingency plans.

CHECK OF UNDERSTANDING

1. What are the two types of risks that have to be considered in testing?
2. Compare and contrast these two risk types.
3. How early in the life cycle can risk impact the testing approach?
4. What does MISRA determine when the level of risk is understood?

TEST ORGANISATION

Test organisation and independence

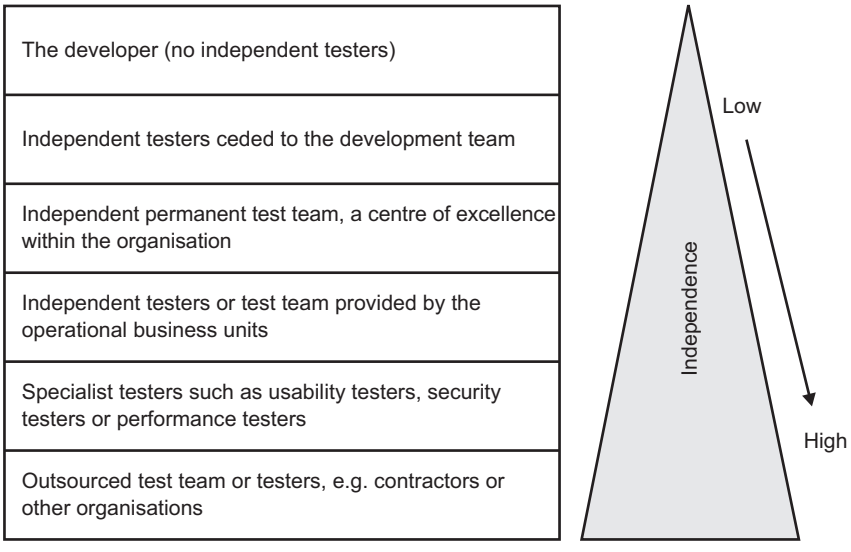
Independent testing is testing carried out by someone other than the creator (developer) of the item being tested. By remaining independent, it is possible to improve the effectiveness of testing if implemented correctly.

As humans we are all capable of making mistakes, from the simplest misspelling or wrong use of syntax to fundamental errors at the core of any documents we write. The problem is that as authors we are less able to see our own errors than someone else, who is less directly associated with the document, would be. This is a problem that is made worse, in the world of software development, by the differing 'world view' of testers and developers. A developer, as the creator and owner of documents and code related to development, perceives these deliverables as being correct when they are delivered. The general awareness that we all make mistakes is, at this stage, overridden by the belief that what has been produced is what is required. A tester, by contrast, will take the view that anything delivered for testing is likely to contain errors and will search diligently to identify and locate those errors.

This is where independent testing is important because it is genuinely hard for authors to identify their own errors, but it is easier for others to see them. There are many options for many levels of independence. In general, the more remote a tester is from the production of the item, the greater is the level of independence. Figure 5.1 indicates the most common roles and the levels of independence they bring.

Of course, independence comes at a price. The greater the level of independence, the greater the likelihood of errors in testing arising from unfamiliarity. Levels of

Figure 5.1 Levels of independent testing



independence will also depend on the size of the organisation. In smaller organisations where everybody contributes to every activity, it is harder to differentiate the role of the tester from any other role, and therefore testers may not be very independent at all. The key in these circumstances is for the testers to have independence of mind, not necessarily to be in an independent (separate) team. In organisations where there are clearly defined roles, it is a lot easier for a tester to remain independent.

It is also possible to mix and match the levels of independence; for example, a test team made up of permanent resources, business unit resources and contractors. For large, complex or safety-critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers.

The Agile approach to development challenges the traditional approach to independence. In this approach everybody takes on multiple roles, so maintaining total independence is not always possible. A tester in this situation has to be able to switch to an independent view, at the relevant points in the project. Testers achieve this independence of view by not assuming anything and by not starting to own the software in the way that a developer might; for example, taking the view that the way the software works is the way it was developed to work.

Independence in the implementation of testing has some key benefits and drawbacks, as in Table 5.1.

Table 5.1 Features of independent testing

Benefits	Drawbacks
• Independent testers are likely to recognise different kinds of failures compared to developers, because of the different backgrounds, technical perspectives and biases. • The tester can see what has been built rather than what the developer thought had been built. • The tester verifies, challenges or disproves assumptions made by stakeholders during specification and implementation of the system.	• Isolation from the development team leading to a lack of collaboration, delays in providing feedback to the development team, or an adversarial relationship between test and development. • Developers lose a sense of responsibility for quality because it may be assumed that they need not worry about errors as the independent test team will find them. • Independent testers can be seen as a bottleneck or to blame for delays in releases. • Independent testers may lack some of the important information (e.g. about the test object), as they have no connection to the stakeholders or developers.

CHECK OF UNDERSTANDING

1. Why is independent testing more effective at finding errors than simply allowing the developer and author to test their own product?
2. Name two benefits of independence.
3. Which organisation provides the lowest level of independence and which provides the highest?

Tasks of a test manager and tester

Test tasks are traditionally carried out by people who make testing a career; however, test tasks may also be carried out by non-testers such as a project manager, quality manager, developer, business and domain expert, infrastructure personnel or IT operations. The availability of resources usually determines the resource types that are deployed on each project; for example, if there are no career testers available an organisation may identify non-testing IT or business resources to carry out the role of tester for a specific project or time period.

The syllabus defines two testing roles: the test manager and the tester. Other roles may exist in your organisation, but they are not covered here.

The testing roles can be undertaken by anyone with the required skills or anyone who is given the right training. For example, the role of a test manager could be undertaken by a project manager. The decision as to who does what will depend on how a project or organisation is structured, as well as the size and number of resources working on a given project.

It is important to understand here the difference between a testing role and a testing job. A role is an activity, or a series of activities, given to a person to fulfil; for example, the role of test manager. A person may therefore have more than one role at any moment depending on their experience and the level of workload on a project. A job is effectively what an individual is employed to do, so one or many roles could make up a job. For example, a test manager could also be a tester.

The tasks undertaken by a test manager align very closely with those undertaken by a project manager and align closely with standard approaches to project management. In this context a test manager is anyone who leads a team of testers (be that one or many testers). Test managers are also known as test programme managers, test team leaders and test coordinators.

Typical test manager tasks may include:

- Coordinating or developing the test policy and test strategy for the organisation.
- Planning the test activities by considering the context and understanding the test objectives and risks. This may include selecting test approaches, estimating test time, effort and cost, acquiring resources, defining test levels and test cycles, and planning defect management.
- Writing and updating test plan(s).
- Coordinating the test plan(s) with project managers, product owners and others.
- Sharing test perspectives with other project activities, such as the code integration planning.
- Initiating the analysis, design, implementation and execution of tests, monitoring test progress and results, and checking the status of execution criteria (or definition of 'done').
- Preparing and delivering test progress reports and test summary reports based on the information gathered.
- Adapting planning based on test results and progress; for example if more defects than planned are found, this will impact the time taken to complete testing and so action will need to be taken to realign the plan.
- Supporting the setting up of the defect management system and adequate configuration management of testware.
- Introducing suitable metrics for measuring test progress and evaluating the quality of the testing and the product.
- Supporting the selection and implementation of tools to support the test process, including budget, and the allocation of time for the effort required to build and support tools.

- Deciding about the implementation of test environment(s).
- Promoting and advocating the tester, the test team, and the test profession within the organisation.
- Developing the skills and careers of testers through training plans, performance evaluations, coaching and so on.

These tasks are not, however, all of the tasks that could be carried out by test managers, just the most common ones. In fact, other resources could take on one or more of these tasks as required, or they may be delegated to other resources by the test manager. In Agile development, some of the above tasks will be handled by the Agile team, especially with reporting. The key is to ensure that everyone is aware of who is doing what tasks, that they are completed on time and within budget, and that they are tracked through to completion.

The other role covered by the syllabus is that of the tester, also known as test analyst or test executor.

The tasks typically undertaken by a tester may include:

- Reviewing and contributing to test plans.
- Analysing, reviewing and assessing user requirements, user stories and acceptance criteria, specifications and models for testability.
- Creating test specifications from the test basis; for example test conditions, and the traceability between test cases, test conditions and the test basis.
- Setting up the test environment (often coordinating with system administration and network management). In some organisations the setting up and management of the test environment could be centrally controlled; in this situation a tester would directly liaise with the environment management to ensure that the test environment is delivered on time and to specification.
- Designing and implementing test cases and test procedures.
- Preparing and acquiring/copying/creating test data.
- Executing tests on all test levels, logging the tests, evaluating the results and documenting the deviations from expected results as defects.
- Using test administration, or management and test monitoring tools as required.
- Automating tests (may be supported by a developer or a test automation expert).
- Evaluating non-functional characteristics such as performance efficiency, reliability and usability.
- Reviewing tests developed by other testers.

As mentioned earlier, the thing to remember when looking at roles and tasks within a test project is that one person may have more than one role and carry out some or all of the tasks applicable to the role. This is different to having a 'job': a 'job' may contain many roles and tasks.

CHECK OF UNDERSTANDING

1. What other names are given to the test manager role?
2. Detail five possible tasks of a test manager.
3. Detail five possible tasks of a tester.
4. Describe the differences between a test manager role and a test manager task.

TEST STRATEGY AND TEST APPROACHES

The test strategy will define how testing will be implemented either in a project or company-wide. It can be:

- developed early in the life cycle, which is known as preventative – in this approach the test design process is initiated as early as possible in the life cycle to stop defects being built into the final solution;
- left until just before the start of test execution, which is known as reactive – this is where testing is the last development stage and is not started until after design and coding have been completed (sometimes it is identified as the waterfall approach, i.e. all development stages are sequential, the next not starting until the previous one has nearly finished).

There are many strategies that can be employed, and they may include:

- Analytical strategies rely on the analysis of some factor such as risk-based testing, where testing is directed to areas of greatest risk (see earlier in this chapter for an overview of risk-based testing).
- Model-based strategies base tests on a model such as statistical information about failure rates (such as reliability growth models) or usage models (such as operational profiles).
- Methodical strategies rely on the systematic use of some predefined tests or test conditions such as failure based (including error guessing and fault attacks), checklist based and quality-characteristic based.
- Process-compliant (or standard-compliant) strategies adhere to the processes developed for use with standards (see ISO/IEC/IEEE 29119-2 or MISRA) and various Agile or traditional waterfall approaches.
- Reactive (or dynamic and heuristic) strategies, such as exploratory testing where testing is more reactive to events than pre-planned, and where execution and evaluation are concurrent tasks.
- Directed (or consultative) strategies, such as those where test coverage is driven primarily by the advice and guidance of technology and/or business domain experts outside or within the test team.

- Regression-averse strategies are designed with the desire to avoid regression of existing capabilities such as those that include reuse of existing test material, extensive automation of functional regression tests and standard test suites.

Different strategies may be combined if required. The decision as to how and why they will be combined will depend on the circumstances prevalent in a project at the time. For example, an organisation may as a standard use an Agile method, but in a particular situation the structure of the test effort could use a risk-based approach to ensure that the testing is correctly focused.

A test strategy will contain generalised descriptions of the test processes to be used; the test approach provides tailoring of the strategy for a particular project or projects. A test approach includes all of the decisions made on how testing should be implemented, based on the (test) project goals and objectives, as well as the risk assessment. It forms the starting point for test planning, selecting the test design techniques and test types to be employed. It should also define the software under test and test entry and exit criteria, often called the definition of 'done' in Agile projects.

The selected approach will depend on the context within which the test team is working, and may consider risks, hazards and safety, available resources and skills, the technology, the nature of the system (e.g. custom built versus COTS), test objectives and regulations.

CHECK OF UNDERSTANDING

1. Name and explain five approaches to the development of the test approach or test strategy.
2. Name one of the standards referred to that dictate the test approach.
3. Can discretion be used when defining a test approach and, if so, what can influence the decision as to which way to approach testing?

TEST PLANNING AND ESTIMATION

Test planning

Test planning is the most important activity undertaken by a test manager in any test project. It ensures that there is initially a list of tasks and milestones in a baseline plan to track progress against, as well as defining the shape and size of the test effort. Test planning is used in development and implementation projects (sometimes called 'greenfield') as well as maintenance (change and fix) activities.

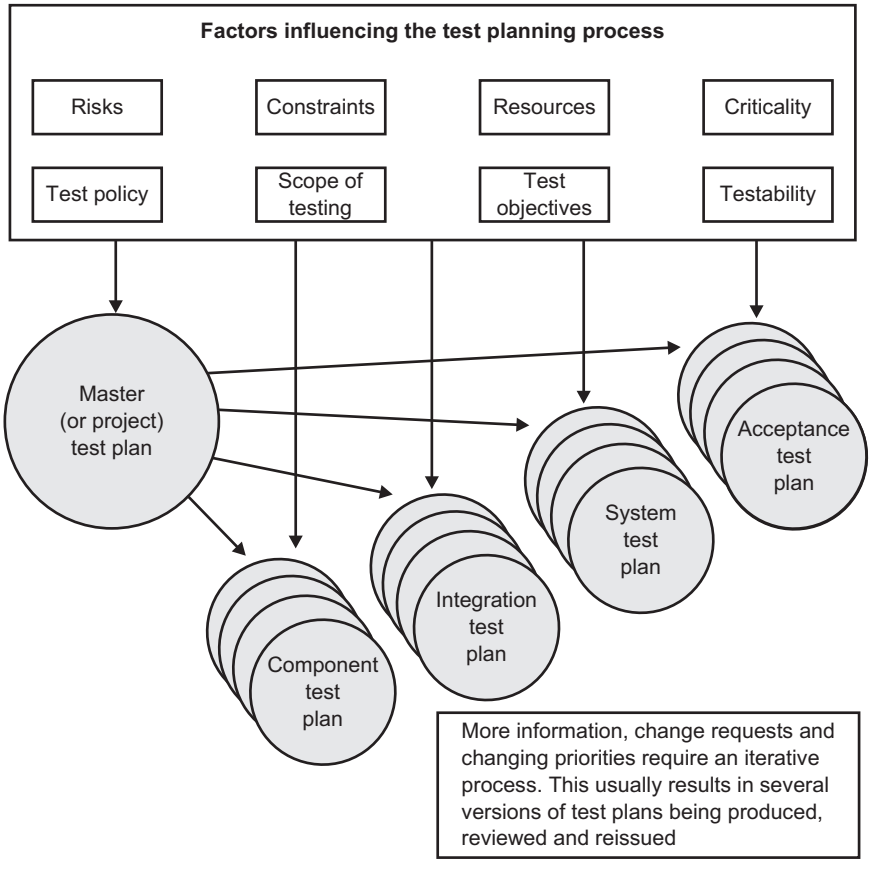
The main document produced in test planning is often called a master test plan or a project test plan. This document defines at a high level the test activities being planned. It is normally produced during the early phases of the project (e.g. initiation) and updated as required via change control as the project develops. It will provide sufficient

information to enable a test project to be established (bearing in mind that at this point in a project little more than requirements may be available from which to plan).

The details of the test-level activities are documented within test-level plans, for example the system test plan. These documents will contain the detailed activities and estimates for the relevant test level.

Figure 5.2 shows where test-level test plans fit into the V model. It shows how a test plan exists for each test level and that they will usually refer to the master test plan.

Figure 5.2 Test plans in the V model



The contents sections of a test plan for either the master test plan or test-level plans are normally identical or very similar. ISO/IEC/IEEE 29119-3, the Software Testing test document standard, contains details of what the content of the plans should be.

The ISO 29119-3 Software Testing Standard identifies that there should be a minimum of 15 sections present in a test plan, as in Table 5.2.

Test planning is a continual activity that spans the life of the test project; it takes place in all life cycle stages. As risks and changes occur, the plan and planning should be amended to recognise these and reflect the current position. The plans will have been baselined (locked down) after initial sign-off, so these changes would normally be managed by the project change process. Baselining a document effectively secures it from further change unless authorised via a change control process.

Table 5.2 Test plan sections

Section no.	Heading	Details
1	Overview	Identifies the document and describes the origins and history
2	Unique identification of the document	The specific unique identifier allocated to this document, e.g. TPR 00001
3	Issuing organisation	Specifies who is responsible for completion and distribution of the document
4	Approval authority	Identifies who is responsible for reviewing and signing off the document before it is issued
5	Change history	A record of each version of the document and any changes that were included for each version
6	Introduction	Explanatory information about the content and structure of the document
7	Scope	Defines the areas of coverage included within the document, test activities, etc.
8	References	Lists referenced documents and identifies repositories for system, software and test information. The references may be separated into 'external' references that are imposed from outside the organisation and 'internal' references that are imposed from within the organisation
9	Glossary	A glossary that defines the terms, abbreviations and acronyms, if any, used in the document

(Continued)

Table 5.2 (Continued)

Section no.	Heading	Details
10	Context of testing	Includes: • Details of the project or sub-processes covered by the plan • Test items – items to be tested • Test scope – what is and isn't included within the scope of testing • Assumptions and constraints • Who the stakeholders are • Testing communication lines inside and outside the test team
11	Risk register	A list of the project and product risks
12	Test strategy	Describes the test approach in the following subsections: • Test sub-process – what test sub-process will be conducted • Test deliverables – documents that will be delivered by the test project • Test design techniques to be used and when • Test completion criteria (exit and entry criteria) • Metrics to be collected • Test data requirements • Test environment requirements • Retesting and regression testing approach • Suspension criteria • Deviations from the organisational test strategy
13	Testing activities and estimates	Documents the activities to be undertaken during testing and the estimate of time required to complete those activities
14	Staffing	Details of the staffing requirements to complete the test plan; this will include roles and responsibilities, hiring needs and any training requirements
15	Schedule	Testing milestones

Test-planning activities

During test planning various activities for an entire system or a part of a system have to be undertaken by those working on the plan. They include:

- Working with the project manager and subject matter experts to determine the scope and the risks that need to be tested. Also identifying and agreeing the objectives of the testing, be they time, quality or cost focused, or a mixture of all three. The objectives will enable the test project to know when it has finished – has time or money run out, or has the right level of quality been met?
- Understanding what delivery model is to be used (waterfall, iterative, Agile, etc.) and defining the overall approach of testing (sometimes called the test strategy) based on this, ensuring that the test levels and entry and exit criteria are defined.
- Liaising with the project manager and making sure that the testing activities have been included within the software life cycle activities such as:
  - design – the development of the software design;
  - development – the building of the code;
  - implementation – the activities surrounding implementation into a live environment.
- Working with the project to decide what needs to be tested, what roles are involved and who will perform the test activities, planning when and how the test activities should be done, deciding how the test results will be evaluated, and defining when to stop testing (exit criteria).
- Building a plan that identifies when and who will undertake the test analysis and design activities. In addition to the analysis and design activities test planning should also document the schedule for test implementation, execution and evaluation. The plan can either be sequential; for example particular dates are defined, or iterative, where the context of each iteration will need to be considered.
- Deciding what the documentation for the test project will be; for example, which plans, how the test cases will be documented and so on.
- Defining the management information, including the metrics required, and putting in place the processes to monitor and control test preparation and execution, defect resolution and risk issues.
- Ensuring that the test documentation generates repeatable test assets; for example, test cases.

ENTRY CRITERIA AND EXIT CRITERIA (DEFINITION OF 'READY' OR DEFINITION OF 'DONE')

Entry criteria are used to determine when a given test activity can start. This could include the planning, when test design and/or when test execution for each level of testing is ready to start. Entry criteria (also known as definition of 'ready' in Agile projects) define the preconditions for undertaking a test activity.

Examples of some typical entry criteria to test execution (for example) may include:

- Availability of testable requirements, user stories or models.
- Test environment available and ready for use (it functions).
- Test tools installed in the environment are ready for use.
- Testable code is available.
- All test data is available and correct.
- All previous test activity has completed and met its exit criteria.

Exit criteria are used to determine when a given test activity has been completed or when it should stop, typically called the definition of 'done' in an Agile project. Exit criteria can be defined for all of the test activities, such as planning, specification and execution as a whole, or to a specific test level for test specification as well as execution.

Exit criteria should be included in the relevant test plans.

Some typical exit criteria might be:

- All tests planned have been executed.
- A certain level of coverage has been achieved.
- The number of unresolved defects is within an agreed limit.
- All high-risk areas have been fully tested, with only minor residual risks left outstanding.
- Cost – when the budget has been spent.
- The number of estimated remaining defects is sufficiently low.
- The evaluated level of quality criteria, such as reliability and performance, is sufficient.
- The schedule has been achieved; for example, the release date has been reached and the product has to go live. This was the case with the millennium testing (it had to be completed before midnight on 31 December 1999), and is often the case with government legislation.

Exit criteria should have been agreed as early as possible in the life cycle; however, they can be, and often are, subject to controlled change as the detail of the project becomes better understood and therefore the ability to meet the criteria is better understood by those responsible for delivery.

CHECK OF UNDERSTANDING

1. What is the international standard for testing called?
2. Identify the 15 sections of the test plan.
3. What activities are contained within test planning?
4. Detail four typical exit criteria.

TEST EXECUTION SCHEDULE

Having developed various test cases and test procedures (including any automated test procedures), which have been assembled into test suites, the test suites can be arranged into a test execution schedule. A test execution schedule documents what test suite will be run in what order and on what day. The order of the execution of test suites will be determined by many things such as prioritisation, processing dependencies; for example suite 1 has to run before suite 7 can be run, whether there are confirmation and regression tests, and finally in the most efficient sequence possible.

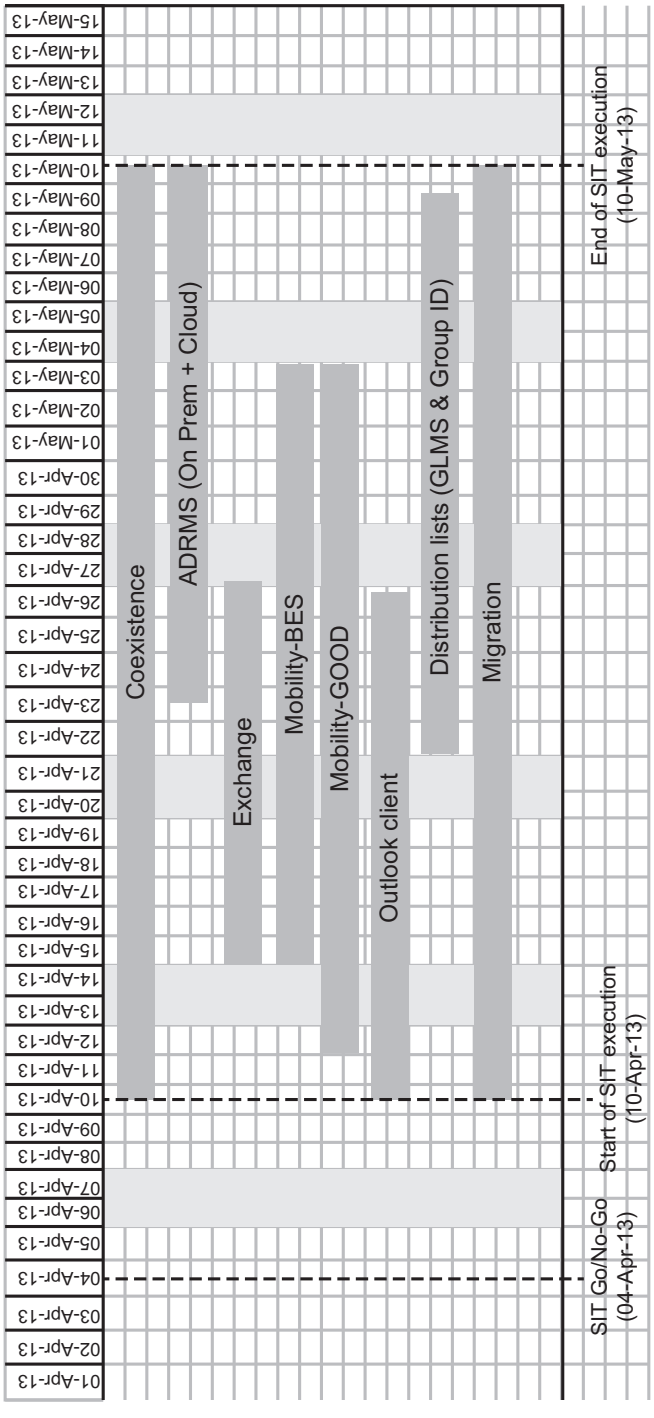
Figure 5.3 (page 175) reflects a high-level test execution schedule for the various system components of a Microsoft Outlook migration.

FACTORS INFLUENCING THE TEST EFFORT

Many things affect the level of effort required to fulfil the test-related aspects of a project to ensure that the objectives of the project, release or iteration are met. These can be split into four main categories, as shown below.

1. Product characteristics:
   - The risks associated with the product (defined during risk-based testing).
   - The quality of the test basis; for example requirements, user stories and so on.
   - The complexity of the product domain.
   - The number of quality characteristics such as reliability.
   - The number of non-functional requirements.
   - The security requirements (perhaps meeting ISO 27001, the security standard).
   - How much documentation is required (e.g. some legislation-driven changes demand a certain level of documentation that may be more than an organisation would normally produce).
   - Requirements for legal or regulatory compliance.
2. Development process characteristics:
   - The stability and maturity of the organisation; for example a very process mature organisation will take a lot less time to achieve what an immature (seat of their pants) organisation would take, as they are likely to make less mistakes.

Figure 5.3 A high-level test execution schedule



- The development model in use, such as Agile or sequential.
- The agreed test approach.
- The tools in use, automation, test management and so on.
- The test process defined in the test strategy and approach.
- Timescales.

3. People characteristics
   - The skills of those involved in the testing and development activity (the lower the skill level in development, the more defects could be introduced, and the lower the skill level in testing, the more detailed the test documentation needs to be).
   - Team cohesion and leadership.
4. Test results:
   - The number and severity of defects expected to be found.
   - The amount of rework needed.

Test estimation

There are many approaches to test estimation: two of the most used are metrics-based and expert-based. The two approaches are quite different, the former being based on data while the latter is a somewhat subjective approach.

The metrics-based approach

This approach relies on data collected from previous or similar projects. This kind of data might include:

- the number of test conditions;
- the number of test cases written;
- the number of test cases executed;
- the time taken to develop test cases;
- the time taken to run test cases;
- the number of defects found;
- the number of environment outages and how long on average each one lasted.

With this approach and the right data, it is possible to estimate quite accurately what the cost and time required for a similar project would be.

It is important that the actual costs and time for testing are accurately recorded. These can then be used to revalidate and possibly update the metrics for use on the next similar project.

The expert-based approach

This alternative approach to metrics is to use the experience of owners of the relevant tasks or experts to derive an estimate (this is also known as the Wide Band Delphi approach). In this context, 'experts' could be:

- business experts;
- test process consultants;
- developers;
- technical architects;
- analysts and designers;
- anyone with knowledge of the application to be tested or the tasks involved in the process.

There are many ways that this approach could be used. Here are two examples:

- Distribute a requirement specification to the task owners and get them to estimate their task in isolation. Amalgamate the individual estimates when received and build in any required contingency, to arrive at the estimate.
- Distribute a requirement specification to known experts who develop their individual view of the overall estimate and then meet together to agree on and/or debate the estimate that will go forward.

Expert estimating can use either of the above approaches individually or mix and match them as required.

Taking all of this into account, once the estimate is developed and agreed, the test manager can set about identifying the required resources and building the detailed plan.

CHECK OF UNDERSTANDING

1. Compare and contrast the two approaches to developing estimates.
2. Provide three examples of what a metrics approach to estimates would use as a base.
3. Name three areas that affect the level of effort to complete the test activity.

TEST MONITORING AND CONTROL

Having developed the test plan, the activities and timescales determined within the test execution schedule need to be constantly reviewed against what is actually happening. This is test monitoring. The purpose of test monitoring is to provide feedback and visibility of the progress of test activities.

The data required to monitor progress can be collected manually; for example, counting test cases developed at the end of each day, or, with the advent of sophisticated test management tools, it is also possible to collect the data as an automatic output from a tool either already formatted into a report, or as a data file that can be manipulated to present a picture of progress.

The progress data is also used to measure exit criteria such as test coverage; for example, 50 per cent requirements coverage achieved.

Having implemented test monitoring to understand progress through the test plan, test control is the corrective action undertaken for issues identified through test monitoring. Slippage of test activity dates or delays in delivery of external components are two potential issue areas. Test-control actions could include:

- reprioritising tests if, for example, software is delivered late (a potential risk);
- changing the test schedule;
- re-evaluating the entry/exit criteria;
- changing the scope of the test activity.

The following test-control activities are likely to be outside the test manager's responsibility. However, this should not stop the test manager making a recommendation to the project manager:

- descoping of functionality; that is, removing some less important planned deliverables from the initial delivered solution to reduce the time and effort required to achieve that solution;
- delaying release into the production environment until exit criteria have been met;
- continuing testing after delivery into the production environment so that defects are found before they occur in production.

Metrics used in testing

In any project, metrics can be collected at any time – either during or at the end of the project, in order to assess:

- progress against the plan, both in terms of activities and budget;
- current quality of the item under test (test object);
- adequacy of the test approach (will it enable all testing to be completed?);
- effectiveness of the test activities with respect to the test objectives.

Common test metrics in use include:

- percentage of planned work done in test case preparation (or percentage of planned test cases prepared);
- percentage of planned work done in test environment preparation;

- test case execution (e.g. number of test cases run/not run, and test cases passed/failed);
- defect information (e.g. defect density, defects found and fixed, failure rate and retest results);
- test coverage of requirements, risks or code;
- subjective confidence of testers in the product;
- task completion, resource allocation and usage, and effort;
- dates of test milestones;
- testing costs, including the cost compared with the benefit of finding the next defect or running the next test.

Ultimately, test metrics are used to track progress towards the completion of testing, which is determined by the exit criteria. So, test metrics should relate directly to the exit criteria.

There is a trend towards 'dashboards', which reflect all of the relevant metrics on a single screen or page, ensuring maximum impact. For a dashboard, and generally when delivering metrics, it is best to use a relatively small but impact-worthy subset of the various metric options available. This is because the readers do not want to wade through lots of data for the key item of information they are after, which invariably is 'Are we on target to complete on time?'

These metrics are often displayed in graphical form, examples of which are shown in Figure 5.4. This reflects progress on the running of test cases and reports on defects found. There is also a box at the top left for some commentary on progress to be documented (this could simply be the issues and/or successes of the previous reporting period).

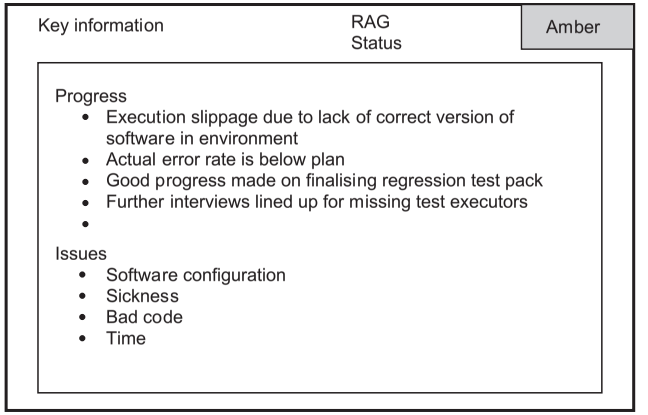
The graph in Figure 5.5 is the one shown at the bottom left of the dashboard in Figure 5.4. It reports the number of defects raised, and also shows the planned and actual numbers of defects.

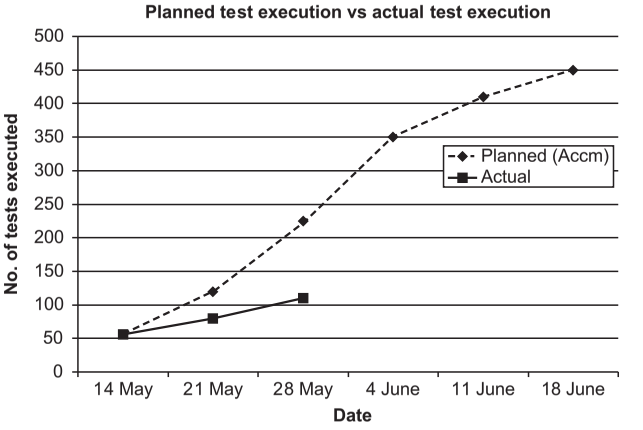
Test reporting

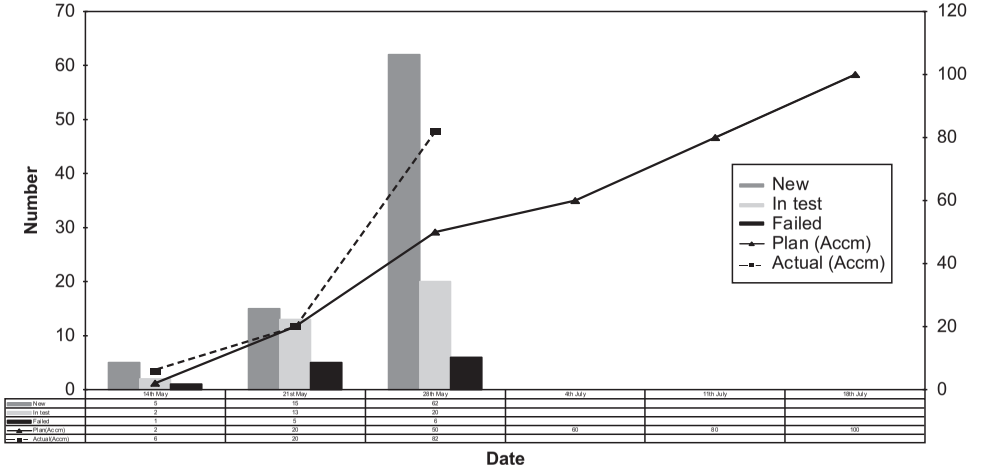
Test reporting is the process whereby test metrics are reported in a summarised format both during and at the end of a test activity, to update the reader regarding the testing tasks undertaken. Test reports produced during the test activity are referred to as test progress reports, whereas a test report produced after a test activity has completed may be referred to as a test summary report.

The test manager regularly issues a test progress report during test monitoring and control for the project stakeholders such as the sponsor, project and programme managers and any product owners. When exit criteria have been met and a test activity completes, the test manager issues a test summary report. This report provides an overview of the test activity undertaken, using data derived from the test progress reports.

Figure 5.4 iTesting executive dashboard







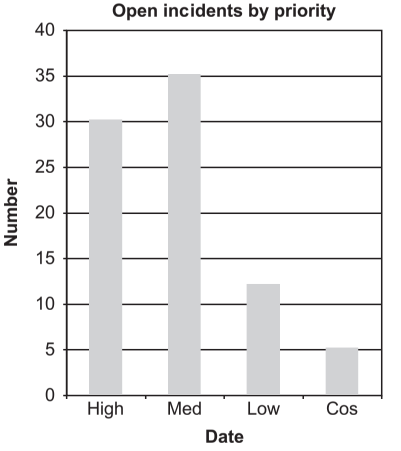
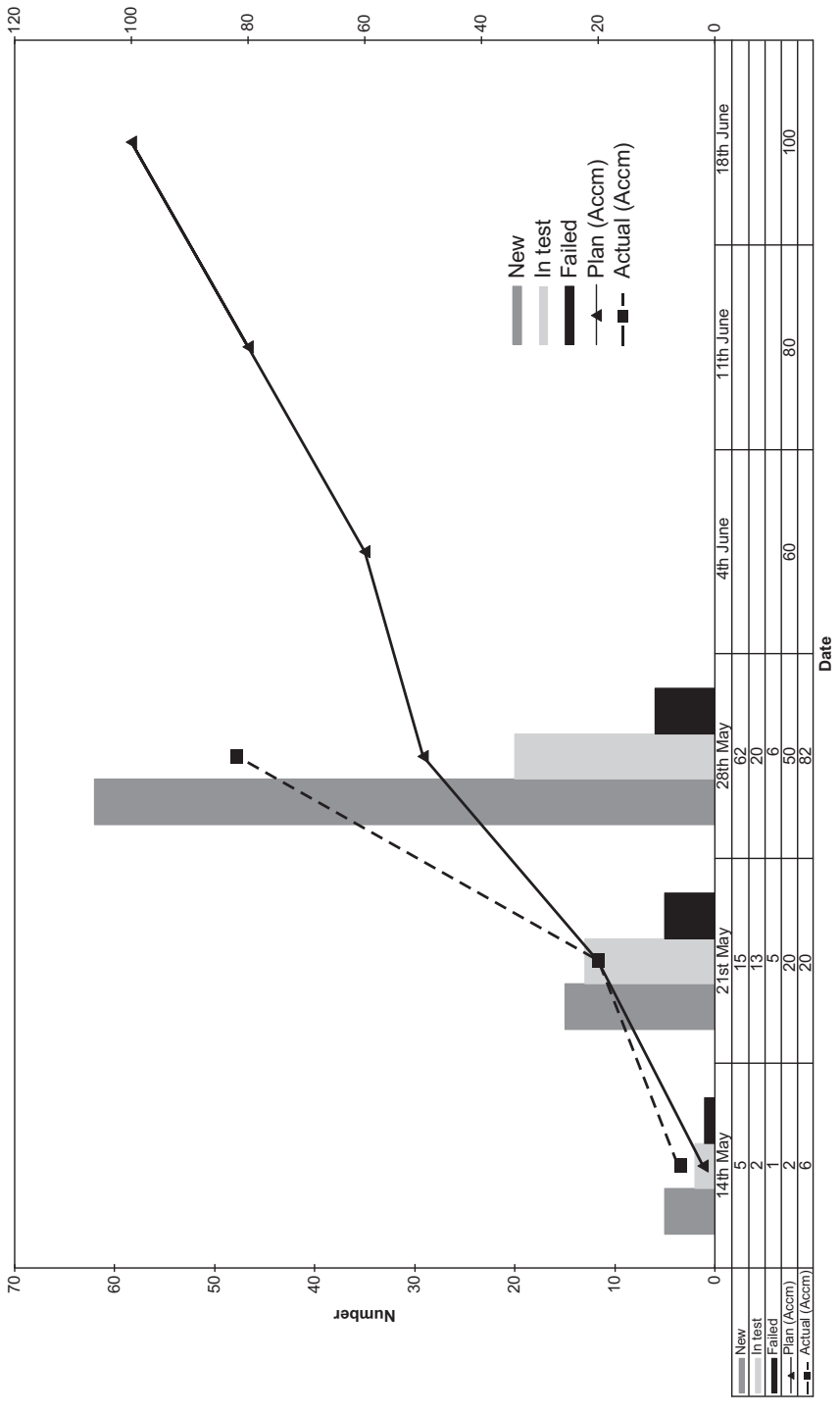


Figure 5.5 Incidents planned/raised



The following information may be included in both a test progress report and a test summary report:

- summary of testing performed;
- information on what occurred during a test period;
- any deviations from the plan, such as schedule changes;
- the status of testing and product quality, relating to either the exit criteria or the definition of 'done';
- blocking factors that have impacted the test schedule;
- metrics reflecting defects, test cases, test coverage, activity progress and resource consumption;
- residual risk;
- reusable test work products produced.

In addition to the above, test progress reports may also include:

- the status of test activities and progress against the test plan;
- factors impacting progress;
- testing planned for the next reporting period;
- the quality of the test object.

Key for any report is that it is focused on the information required based upon the report's audience; some recipients may require graphs, while others wish to see the detailed data. It is the responsibility of the test manager to ensure that the recipient requirements for reports are understood before any test activity starts.

In an Agile project, the test progress reporting may be included in task boards, effect summaries and burn-down charts, which may also be discussed in the daily stand-up meeting, where the project team review progress during the previous period (often a day) and what is planned for the next period.

ISO 29119-3 documents required contents for both test progress reports (called test status report in the standard) and a test summary report (called a test completion report in the standard).

Tables 5.3a and 5.3b detail the two separate standard contents.

The information gathered can also be used to help with any process improvement opportunities. This information can be used to assess whether:

- the goals for testing were correctly set (where they achievable; if not why not?);
- the test approach or strategy was adequate (e.g. did it ensure there was enough coverage?);
- the testing was effective in ensuring that the objectives of testing were met.

Table 5.3a Test progress report outline

Section no.	Heading	Details
1	Overview	Identifies the document and describes the origins and history
2	Unique identification of the document	The specific unique identifier allocated to this document, e.g. TP 00001
3	Issuing organisation	Specifies who is responsible for completion and distribution of the document
4	Approval authority	Identifies who is responsible for reviewing and signing off the document before it is issued
5	Change history	A record of each version of the document and any changes that were included for each version
6	Introduction	Explanatory information about the content and structure of the document
7	Scope	Defines the areas of coverage included within the document, test activities etc.
8	References	Lists referenced documents and identifies repositories for system, software and test information. The references may be separated into 'external' references that are imposed from outside the organisation and 'internal' references that are imposed from within the organisation
9	Glossary	A glossary that defines the terms, abbreviations and acronyms, if any, used in the document
10	Test status	Includes: • Reporting period • Progress against the test plan • Factors blocking progress • Test measures • New and changed test risks • Testing planned in the next period

Table 5.3b Test summary report outline

Section no.	Heading	Details
1	Overview	Identifies the document and describes the origins and history
2	Unique identification of the document	The specific unique identifier allocated to this document, e.g. TSR 00001
3	Issuing organisation	Specifies who is responsible for completion and distribution of the document
4	Approval authority	Identifies who is responsible for reviewing and signing off the document before it is issued
5	Change history	A record of each version of the document and any changes that were included for each version
6	Introduction	Explanatory information about the content and structure of the document
7	Scope	Defines the areas of coverage included within the document, test activities, etc.
8	References	Lists referenced documents and identifies repositories for system, software and test information. The references may be separated into 'external' references that are imposed from outside the organisation and 'internal' references that are imposed from within the organisation
9	Glossary	A glossary that defines the terms, abbreviations and acronyms, if any, used in the document
10	Testing performed	Includes: • A summary of testing performed • Deviations from planned testing • Test completion evaluation, e.g. have exit criteria all been met, if not why not • Factors that blocked progress • The collated test measures • Residual risks • Test deliverables • Reusable test assets • Lessons learnt

CHECK OF UNDERSTANDING

1. Name four common test metrics.
2. Name the 10 headings in the ISO 29119-3 test summary report.
3. Identify three ways a test manager can control testing if there are more tests than there is time to complete them.

DEFECT MANAGEMENT

A defect is any unplanned event occurring that requires further investigation. In testing, this translates into anything where the actual result is different from the expected result. A defect when investigated may be a defect; however, it may also be a change to a specification or an issue with the test being run. It is important that a process exists to track all defects through to closure. This process has to be agreed by all parties involved and can be quite informal.

Defects can be raised at any time throughout the Software Development Life Cycle, from reviews of the test basis (requirements, specifications etc.), coding, static analysis, test specification and dynamic testing.

Typical defect reports have the following objectives:

- To provide developers and other parties with feedback on the problem to enable identification, isolation and correction as necessary. It must be remembered that most developers and other parties who will correct the defect or clear up any confusion will not be present at the point of identification, so without full and concise information they will be unable to understand the problem, and possibly therefore be unable to understand how to go about fixing it. The more information provided, the better.
- To provide test managers with a means of tracking the quality of the system under test and the progress of the testing. Key metrics used to measure progress is a view of how many defects are raised, their priority and finally that they have been corrected and signed off.
- To provide ideas for test process improvement. For each defect the point of injection should be documented, for example a defect in requirements or code, and subsequent process improvement can focus on that particular area to stop the same defect occurring again.

A defect report filed during dynamic testing typically includes:

- an identifier;
- a title or a short summary of the defect being raised;
- date of the defect report, issuing organisation and author;
- identification of the test item and environment being used;

- the development life cycle phase it was identified in;
- a description of the defect to enable reproduction and resolution;
- expected and actual results;
- scope or degree of impact (severity) of the defect on the stakeholders;
- urgency (priority) to fix;
- state of the defect report; for example is it open, deferred, closed and so on?
- conclusion, recommendations and approvals;
- change history (updates reflecting the sequence of action taken to resolve the defect);
- any references.

Defect management is the process of recognising, investigating, taking action and disposing of defects. It involves recording defects, classifying them and identifying the impact. The process of defect management ensures that defects are tracked from recognition to correction, and finally through retest and closure. It is important that organisations document their defect management process and ensure that they have appointed someone (often called a defect manager/coordinator) to manage/police the process.

Defects are raised on defect reports, either electronically via a defect management system (from Microsoft Excel to sophisticated defect management tools) or on paper.

The syllabus also recognises that ISO 29119-3 defines a test defect report (called a test defect report) which has sections aligned with those documented above.

CHECK OF UNDERSTANDING

1. Identify three details that are usually included in a defect report.
2. What is the name of the standard that includes an outline of a test defect report?
3. What is a test defect?

CONFIGURATION MANAGEMENT

The purpose of configuration management is to establish and maintain the integrity of the component or system, the testware and their relationships to one another throughout the project and product life cycle. It involves managing products, facilities and processes by managing the information about them, including changes, and ensuring that they are what they are supposed to be in every case.

For testing, configuration management will involve controlling both the versions of code to be tested and the documents used during the development process; for example, requirements, design and plans.

In both instances, configuration management should ensure that each test item is uniquely identified and provide full traceability throughout the test process; for example, a requirement should be traceable through to the test cases that are run to test its levels of quality and vice versa.

Effective configuration management is important for the test process as the contents of each release of software into a test environment must be understood and at the correct version, otherwise testers could end up wasting time because either they are testing an invalid release of the software or the release does not integrate successfully, leading to the failure of many tests.

In most instances the project will have already established configuration management processes that will define the documents and code to be held under configuration management. If this is not the case, then during test planning the process and tools required to establish the right configuration management processes will need to be selected/implemented by the test manager.

The same principle applies to testware. Each item of testware (such as a test procedure) should have its own version number and be linked to the version of the software it was used to test. For example, test procedure TP123a might be used for software Release A and TP123b might be used for software Release B – even though both have the same purpose and even expected results. However, another test procedure, TP201, may be applicable to all releases.

A good configuration management system will ensure that the testers can identify exactly what code they are testing as well as have control over the test documentation such as test plans, test specification, defect logs and so on.

CHECK OF UNDERSTANDING

1. Define configuration management.
2. What can be stored under configuration management?
3. Why is it important to have effective configuration management?

SUMMARY

In this chapter we have looked at the component parts of test management. We initially explored risk and testing. When developing the test plan, the test manager and tester will look at the product risks (risks that relate directly to the failure of the product in the live environment) to decide what is important to test, as well as ensuring that any project risks (risks relating to the delivery of the project) are mitigated.

The importance of independence in the test organisation and how independence helps to ensure that the right focus is given to the test activity was reviewed. Independence is gained by separating the creative development activity from the test activity and we looked at the different levels of independence that are achievable:

- the developers – low independence;
- independent testers ceded to the development team;
- independent permanent test team, a centre of excellence within the organisation;
- independent testers or test team provided by the operational business unit;
- outsourced test team or the use of independent contractors – high independence.

We have looked at the test strategy and approach and how they shape the test activity based on many influences, including risks and the objectives of the testing.

We have reviewed two roles that exist within a test project: test manager and tester. Both roles are important to the delivery of testing, but could be vested in one or many people; for example, one person could have the role of test manager and tester. A test manager has responsibility for all of the planning activity, while the tester has responsibility for activities that surround the preparation of test cases.

ISO 29119-3 provides outlines of four test-planning documents:

- the test plan;
- the test progress report;
- the test summary report;
- the test defect report.

Test management depends not only on the preparation of the required documents but also on the development of the right entry and exit criteria and estimates, the monitoring of progress through the plan and the control activities implemented to ensure the plan is achieved.

Test estimating can be achieved in one of two ways: metrics or an expert-based approach.

After a plan of activity has been developed and time begins to pass, the test manager needs to monitor the progress of the activities. If any activity is delayed or there has been a change of any kind in the project itself, the test manager may need to revise the plan or take other actions to ensure that the project is delivered on time.

We explored how the defects found during testing are recorded, and we reviewed the level of detail that needs to be recorded to ensure that any defect is fully understood and that any fix then made is the right one.

Finally, we looked at configuration management. When running test cases against the code, it is important that the tester is aware of the version of code being tested and the version of the test being run. Controlling the versioning of the software and test assets

is called configuration management. Lack of configuration management may lead to issues like loss of already-delivered functionality, reappearance of previously corrected errors and no understanding of which version of the test was run against which version of the code.

Example examination questions with answers

E1. K1 question
When assembling a test team to work on an enhancement to an existing system, which of the following has the highest level of test independence?

a. A business analyst who wrote the original requirements for the system.
b. A permanent programmer who reviewed some of the new code but who has not written any of it.
c. A permanent tester who found the most defects in the original system.
d. A contract tester who has never worked for the organisation before.

E2. K2 question
Which of the following correctly identify a metrics-based approach to estimation?

a. Groups of experts provide estimates based on their experience.
b. Volumes of defects identified at a given stage in a project.
c. Records of defects found in a similar stage in another project and the time taken to remove them.
d. Comparison of the estimates given by testers on the project and independent experts.

E3. K2 question
Which of the following is appropriate content for a test summary report?

i. The status of testing and progress against the test plan.
ii. Information about what occurred during a test period.
iii. A review of test activity progress and resource consumption for the system testing phase.
iv. An assessment of the quality of the test object at the present stage of testing.

a. i and ii.
b. ii and iii.
c. iii and iv.
d. i and iv.

E4. K1 question
Which of the following terms is used to describe the management of software components comprising an integrated system?

a. Configuration management.
b. Defect management.
c. Test monitoring.
d. Risk management.

E5. K1 question
A new system is about to be developed. Which of the following functions has the *highest* level of risk?

a. Likelihood of failure = 20%; impact value = £100,000.
b. Likelihood of failure = 10%; impact value = £150,000.
c. Likelihood of failure = 1%; impact value = £500,000.
d. Likelihood of failure = 2%; impact value = £200,000.

E6. K2 question
Which of the following statements about risks is most accurate?

a. Project risks rarely affect product risk.
b. Product risks rarely affect project risk.
c. A risk-based approach is more likely to be used to mitigate product rather than project risks.
d. A risk-based approach is more likely to be used to mitigate project rather than product risks.

Answers to questions in the chapter

SA1. The correct answer is c.
SA2. The correct answer is a.
SA3. The correct answer is a.

Answers to example examination questions

E1. The correct answer is d.
In this scenario, the contract tester who has never worked for the organisation before has the highest level of test independence. The three others are less independent because they are likely to make assumptions based on their previous knowledge of the requirements, code and general functionality of the original system.

Note that independence does not necessarily equate to most useful. In practice, most test or project managers would recruit a permanent tester who has worked on the original system in preference to a contract tester with no knowledge of the system. However, when assembling a team, it is useful to have staff with varying levels of test independence and system knowledge.

E2. The correct answer is c.

a. This approach is known as the Wide Band Delphi estimation technique and is based on multiple, well-informed estimates but not on data, as required by a metrics-based approach.
b. Volumes of defects identified is a valid metric and could be valuable in estimation, but without data about how much effort was required to remove them we would not be able to use the data about volumes in a metrics-based approach.
c. This is a valid metrics-based approach because data about the volume of defects and the time taken to remove them is provided, albeit on a different project.
d. This is potentially an effective way to improve the ability of testers to estimate accurately, but it is expert-based rather than metrics-based because there is no actual data to provide measures of achievement.

E3. The correct answer is b.
While all of the options describe content that summarises activity, option i is focused on identifying specific progress information, that is, whether or not progress is consistent with the plan. Item iv is also specific to progress towards an objective (the quality of the test object). Items ii and ii, in contrast, provide a broader view of what happened and what has been done, that is, they summarise rather than report specific progress. For these reasons, items ii and iii are more appropriate to a test summary report, while items i and iv are more appropriate to a test progress report.

E4. The correct answer is a.
Defect management is the collection and processing of defects raised when errors and defects are discovered. Test monitoring identifies the status of the testing activity on a continual basis. Risk management identifies, analyses and mitigates risks to the project and the product. Configuration management is concerned with the management of changes to software components and their associated documentation and testware.

E5. The correct answer is a.
In b, the product of probability × impact has the value £15,000; in c, the value is £5,000 and in d, it is £4,000. The value of £20,000 in a is therefore the highest.

E6. The correct answer is c.
In general, project risk and product risk can be hard to differentiate. Anything that impacts on the quality of the delivered system is likely to lead to delays or increased costs as the problem is tackled. Anything causing delays to the project is likely to threaten the delivered system's quality. The risk-based approach is an approach to managing product risk through testing, so it impacts most directly on product risk.

6 TOOL SUPPORT FOR TESTING

Peter Williams

INTRODUCTION

As seen in earlier chapters there are many tasks and activities that need to be performed during the testing process. In addition, other tasks need to be performed to support the testing process.

In order to assist in making the testing process easier to perform and manage, many different types of test tools have been developed and used for a wide variety of testing tasks. Some of them have been developed in-house by an organisation's own software development or testing department. Others have been developed by software houses (also known as test-tool vendors) to sell to organisations that perform testing. More recently, open source tools have been developed that can be reused and enhanced. Even within the same type of tool, some will be home-grown while others will be developed as open source tools or by test-tool vendors.

This chapter discusses the potential benefits and pitfalls associated with test tools in general. It then describes the most commonly used types of test tools and concludes with a process for introducing a tool into a test organisation.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions that follow the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction** (page 2).

Test tool considerations (K2)

- FL-6.1.1 Classify test tools according to their purpose and the test activities they support.

- FL-6.1.2 Identify benefits and risks of test automation. (K1)
- FL-6.1.3 Remember special considerations for test execution and test management tools. (K1)

Effective use of tools (K1)

- FL-6.2.1 Identify the main principles for selecting a tool.
- FL-6.2.2 Recall the objectives for using pilot projects to introduce tools.
- FL-6.2.3 Identify the success factors for evaluation, implementation, deployment, and on-going support of test tools in an organization.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K2)
Which of the following pairs of test tools are *likely* to be *most useful* during the test analysis stage of the test process?

i. Test execution tool.
ii. Test data preparation tool.
iii. Test management tool.
iv. Requirements management tool.

a. i and ii.
b. i and iv.
c. ii and iii.
d. iii and iv.

Question SA2 (K1)
Which of the following is *most likely* to cause failure in the implementation of a test tool?

a. Underestimating the demand for a tool.
b. The purchase price of the tool.
c. No agreed requirements for the tool.
d. The cost of resources to implement and maintain the tool.

Question SA3 (K2)
What benefits do static analysis tools have over test execution tools?

a. Static analysis tools find defects earlier in the life cycle.
b. Static analysis tools can be used before code is written.

c. Static analysis tools test that the delivered code meets business requirements.
d. Static analysis tools are particularly effective for regression testing.

WHAT IS A TEST TOOL?

Definition of a test tool

The ISTQB Glossary of Testing Terms defines a test tool as:

> A software product that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution and test analysis.

Therefore, a test tool can be thought of as a piece of software that is used to make the testing process more effective or efficient. In other words, anything that makes testing easier, quicker, more accurate and so on.

This book will focus on those test tools that are listed in the 2018 syllabus. These are listed in Table 6.5 on pages 227 to 232 and are, generally, the test tools that are most commonly used in the testing process. Other test tools that have been removed from the 2018 syllabus (such as Test Comparators) are also discussed to help understand newer tools and for completeness. But they do not need to be studied for the exam and are marked with [Not in Syllabus].

Benefits and risks of using any type of tool

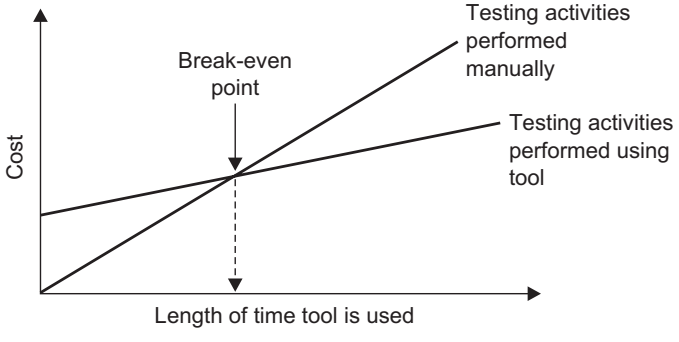
Let us consider the building of a new hotel and examine the similarities with the introduction and use of test tools. Test tools need to be thought of as long-term investments that need maintenance to provide long-term benefits. Similarly, building a hotel requires a lot of upfront planning, effort and investment. Even when the hotel is ready for use, there is still a continual long-term requirement for the provision of services such as catering, cleaning, building maintenance, provision of staff to provide ad hoc services to customers and so on and, from time to time, the need for upgrades to infrastructure to keep up with new technology and customer demands. The long-term benefit is that this upfront investment and ongoing maintenance and support can provide substantial income in return.

In addition, there are risks that, over a period of time, the location of the hotel will become less attractive, resulting in lower demand, lower usage and a maintenance cost that is greater than the income received. Therefore, the initial investment is wasted because the ongoing need/requirement did not exist.

The graph in Figure 6.1 demonstrates a typical payback model for implementing a test execution tool. The same principle applies to the majority of test tools. Note that there is

an ongoing maintenance cost of using the tool, but this ongoing maintenance cost needs to be less than the cost of performing testing activities without the tool if the investment is to be worthwhile.

Figure 6.1 Test tool payback model



The same advantages and disadvantages apply to the use of most types of test tool. However, there are exceptions to this generalisation (and to the same generalisation made in the ISTQB syllabus). Some tools, such as comparators, can be used virtually straight out of the box. A comparator can check whether one large test file is the same as another. If it is different, it can identify and report on the differences. This is very difficult and time-consuming to do manually. In addition, defect management tools are fairly intuitive and easy for both experienced and novice testers to use. They are also likely to provide a 'quick win'.

Other tools can be built by developers in-house as the need arises. For instance, test harnesses, test oracles or test data preparation tools may be relatively easy to produce for developers with a good understanding of the tool requirements and the systems and databases in the test environment. More recently, open source tools have allowed developers and testers to use freeware tools as the building blocks for developing in-house tools to meet specific needs. In addition, test tools have been developed by the UK Financial Conduct Authority for use by banks and building societies that participate in the Faster Payments and Account Switcher schemes.

Benefits

The main benefit of using test tools is similar to the main benefit of automating any process. That is, the amount of time and effort spent performing routine, mundane, repetitive tasks is greatly reduced. For example, consider the time and cost of making consumer goods by hand or in a factory.

This time saved can be used to reduce the costs of testing or it can be used to allow testers to spend more time on the more intellectual tasks of test planning, analysis and design. In turn, this can enable more focused and appropriate testing to be done – rather than having many testers working long hours, running hundreds of tests.

Related to this is the fact that the automation of any process usually results in more predictable and consistent results. Similarly, the use of test tools provides more predictable and consistent results as human failings, such as manual-keying errors, misunderstandings, incorrect assumptions, forgetfulness and so on, are eliminated. It also means that any reports or findings tend to be objective rather than subjective. For instance, humans often assume that something that seems reasonable is correct, when in fact it may not be what the system is supposed to do.

The widespread use of databases to hold the data input, processed or captured by the test tool, means that it is generally much easier and quicker to obtain and present accurate test management information, such as test progress, defects found/fixed and so on (see **Chapter 5**). The introduction of web-based tools that have databases stored in the cloud means that such information is available to global organisations 24 hours per day, seven days per week. This facilitates round the clock working and can reduce elapsed times to analyse, fix and retest defects.

Risks

Most of the risks associated with the use of test tools are concerned with over-optimistic expectations of what the tool can do and a lack of appreciation of the effort required to implement and obtain the benefits that the tool can bring.

For example, consider the production environments of most organisations thinking about using test tools. They are unlikely to have been designed and built with test tools in mind. Therefore, assuming that you want a test environment to be a copy of production (or at least as close to it as possible), you will also have a test environment that is not designed and built with test tools in mind.

Consider the test environments used by vendors to demonstrate their test tools. If you were the vendor, would you design the environment to enable you to demonstrate the tool at its best or to demonstrate the shortcomings it may encounter in a typical test environment?

Therefore, unless detailed analysis and evaluation is done, it is likely that test tools will end up as something that seemed a good idea at the time but have been largely a waste of time and money. A process for avoiding such problems when introducing a tool into an organisation is described later in this chapter.

After a test tool has been implemented and measurable benefits are being achieved, it is important to put in sufficient effort to maintain the tool, the processes surrounding it and the test environment in which it is used. Otherwise there is a risk that the benefits being obtained will decrease and the tool will become redundant. Additionally, opportunities for improving the way in which the tool is used could also be missed.

For example, the acquisition of various test tools from multiple vendors will require interfaces to be built or configured to import and export data between tools. Otherwise much time may be spent manually cutting and pasting data from one tool to another. If this is not done, then data inconsistencies and version control problems are likely to arise. Similar problems may arise when testing with third-party suppliers or as a result of mergers and acquisitions. The increase in common standards for interfaces such as

Extensible Markup Language (XML) means that the capability for developing successful interfaces is greater, but substantial time and effort are often still required.

Maintenance effort will also be required to upgrade and reconfigure tools so that they remain compliant with new platforms or operating systems.

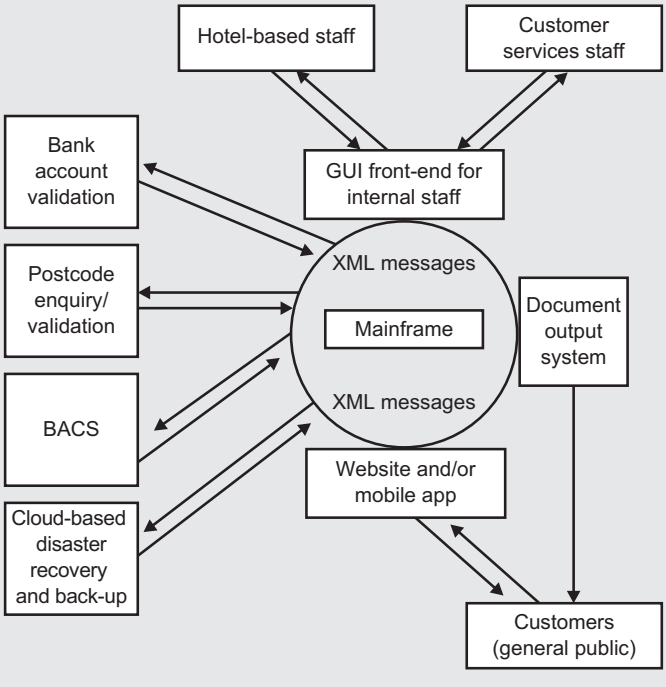
EXAMPLE – HOTEL CHAIN SCENARIO

An example of a hotel chain with several UK-based hotels will be used throughout this chapter. The systems that comprise the organisation's system architecture are shown in Figure 6.2.

The general public can book rooms at any of the chain's hotels by:

- contacting staff in the hotel, who then use a Graphical User Interface (GUI) front-end to make the booking;
- telephoning customer services who then use a GUI front-end to make the booking;
- using the company's website to make the booking online;
- using a mobile app that can be downloaded from the hotel chain's website.

Figure 6.2 Hotel system architecture



In all cases, communication with the mainframe computer is done via a middleware layer of XML messages.

There is a document production system that produces PDF versions of customer correspondence such as booking confirmations, bills, invoices and so on. These are stored securely and can be downloaded by customers from the website.

Direct debit payments are made via Bankers Automated Clearing Services (BACS). Files are transmitted and confirmation and error messages are received back. Credit card payments can be made. An enhancement to the security systems is being made to comply with Payment Card Industry standards. Payments can also be made by leading electronic payment systems (e.g. PayPal).

Validation of bank account details is performed by sending XML messages to and from a third-party system.

Validation and enquiry of address and postcode is also performed by sending XML messages to and from a third-party system.

A new release of the system is planned for six months' time. This will include:

- Code changes to replace the XML middleware layer. Mainframe changes will be performed by an outsourced development team in India.
- Various changes to website screens to improve usability.
- The introduction of a new third-party calendar object from which dates can be selected.
- Removal of the ability for customers to pay by cheque.
- An amended customer bill, plus two other amended documents.
- Two new output documents.
- Fixes to various existing low- and medium-severity defects.
- Improvements to disaster recovery by using cloud-based methods.
- Ongoing enhancements to the mobile app using Agile development methods. These will be deployed to production approximately every three weeks.

CHECK OF UNDERSTANDING

1. Would you expect a quick return on your investment in test tools? Why?
2. Describe three potential benefits of using test tools.
3. Describe two risks of using test tools.

TEST TOOLS

Types of tool

There are several ways in which test tools can be classified. They can be classified according to:

- their purpose;
- the test process and the Software Development Life Cycle with which they are primarily associated;
- the type of testing that they support;
- the source of tool (shareware, open source, free or commercial);
- the technology used;
- who uses them.

In this book, test tools will be classified according to the type of activity they support (as in the ISTQB Foundation Level syllabus).

Tool support for management of testing and testware

Test management tools and application life cycle management tools Test management tools and application life cycle management (ALM) tools provide support for various activities and tasks throughout the testing process. The main difference between a test management tool and an ALM tool is that:

- A test management tool tends to focus on the test process and allow integration to other tools (but this integration may need to be built using APIs).
- An ALM tool typically has built-in integration with other tools.

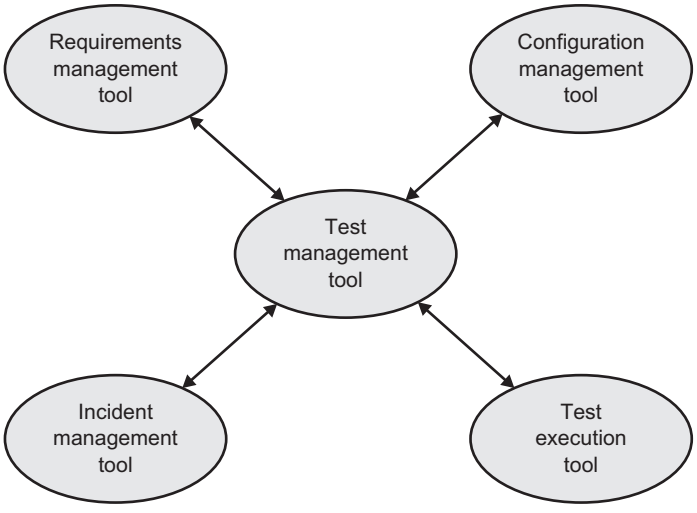
In the remainder of this section any service/function provided by a test management tool will also be met by an ALM tool unless otherwise stated.

Consequently, standalone test management tools tend to be cheaper than ALM tools.

The diagram in Figure 6.3 shows how a test management tool (or ALM tool) is the hub or centre of a set of integrated test tools.

Test management (and ALM) tools provide an architecture for creating, storing and editing test procedures. These may be linked or traced to requirements, test conditions and risks. Such test procedures can then be prioritised or grouped together and scheduled so that they are run in the most effective and efficient order. Some test management tools allow dependencies to be recorded so that tests that will fail owing to a known defect can be highlighted and left unexecuted. This allows testers to be redirected to run the highest priority tests available rather than waste their time and the test data they have prepared on tests that are certain to fail.

Figure 6.3 An integrated set of tools



Tests can be recorded as passed or failed and usually a test management (or ALM) tool provides an interface to a defect management tool so that a defect can be raised if the actual and expected results differ.

Test management (or ALM) tools can provide management information and reports on test procedures passed or failed. The amount of integration with other specialist tools is significant here. For instance, integration with requirements management tools allows reports to be produced on test progress against one or more requirements. Integration with incident management tools also allows reports to include analysis of defects against requirements.

Test management tools generally hold data in a database. This allows a large amount of reports and metrics to be produced. The metrics produced can be used as inputs to:

- test and project management to control the current project;
- estimates for future projects;
- identify weaknesses or inefficiencies in the development or test processes that can be subsequently investigated with the aim of improving them.

Test management information reports should be designed to meet the needs of project managers and other key users. It may be necessary to export/import data in appropriate formats to other tools such as:

- spreadsheets;
- project management/scheduling tools;
- management accounting systems;
- human resources/personnel systems and so on.

A test management (or ALM) tool can also enable reuse of existing testware in future test projects.

USE IN HOTEL CHAIN SCENARIO

In the scenario, a test management (or ALM) tool can be used to write down and store requirements for new functionality and subsequently to hold the test conditions necessary to test these requirements.

It can also be used to record whether tests have passed or failed and to produce test management information on progress to date.

Additionally, requirements and test conditions from previous developments will already exist in the test management tool. These can be used as the basis for the regression testing required. Indeed, a regression test pack may already exist. Clearly the regression test pack would have to be reviewed and amended as necessary to make it relevant to this release. However, the benefit is that much of the previous work could be reused, which, in turn, means that much less effort will be involved to create a regression test pack.

Defect management tools Defect management tools (also known as incident management tools) are one of the most widely used types of test tool. At a basic level, defect management tools are used to perform two critical activities: creation of a defect report; and maintenance of details about the defect as it progresses through the defect life cycle.

The level of detail to be captured about the defect can be varied depending on the characteristics of the tool itself and the way in which the defect management tool is configured and used by the test organisation.

For example, the defect management tool can be configured so that lots of mandatory information is required in order to comply with industry or generic standards such as IEEE 1044. In addition, workflow rules may also be applied to ensure that the agreed defect life cycle is strictly adhered to, with defects only able to be assigned to particular teams or users. Alternatively, the tool can be configured to require very limited mandatory information, with most fields being free format.

Defect management tools also use a database to store and manage details of defects. This allows the defect to be categorised according to the values stored in appropriate fields. Such values will change during the defect life cycle as the defect is analysed, debugged, fixed and retested. It is often possible to view the history of changes made to the defect.

The database structure also enables defects to be searched and analysed (using either filters or more complex Structured Query Language (SQL)-type queries). This provides the basis for management information about defects. Note that as the values held

against each defect change, the management information will also change. Therefore, users need to be aware of the danger of using outdated reports.

This data can be used in conjunction with data held in test management tools when planning and estimating for future projects. It can also be analysed to provide input to test process improvement projects.

Fields in the database structure normally include:

- priority (e.g. high, medium, low);
- severity (e.g. high, medium, low);
- assignee (the person to whom the defect is currently assigned, e.g. a developer for debugging, a tester to perform retesting);
- status in the defect life cycle (e.g. New, Open, Fixed, Reopen, Closed).

This allows management information to be produced from the defect management database about the number of high-priority defects with a status of Open or Reopen that are assigned to, say, Peter Morgan, compared with the number assigned to Brian Hambling.

ALM tools typically include fully integrated defect management tools as part of their core product, while other defect management tools can be integrated with test management, requirements management and/or test execution tools. Such integration enables defects to be input and traced back to test cases and requirements.

USE IN HOTEL CHAIN SCENARIO

A defect management tool can be used to raise new defects and process them through the defect life cycle until resolved. It can also be used to check whether defects (or similar defects) have been raised before, especially for defects raised during regression testing.

A defect management tool could also be used to prioritise defects so that developers fix the most important ones first. It could also highlight clusters of defects. This may suggest that more detailed testing needs to be done on the areas of functionality where most defects are being found, as it is probable that further defects will be found as well.

Requirements management tools Requirements management tools are used by business analysts to record, manage and prioritise the requirements of a system. They can also be used to manage changes to requirements – something that can be a significant problem for testers, as test cases are designed and executed to establish whether the delivered system meets its requirements. Therefore, if requirements change after tests have been written then test cases may also need to change. There is

also a potential problem of changes not being communicated to all interested parties, thus testers could be using an old set of requirements while new ones are being issued to developers.

The use of a traceability function within a requirements tool (and/or integrated with an ALM or test management tool) enables links and references to be made between requirements, functions, test conditions, defects and other testware items. This means that as requirements change, it is easy to identify which other items may need to change.

Some requirements management tools can be integrated with test management tools, while ALM tools typically enable requirements to be input and related to test cases within the ALM tool.

Requirements management tools also enable requirements coverage metrics to be calculated easily, as traceability enables test cases to be mapped to requirements.

As can be seen, traceability can create a lot of maintenance work, but it does highlight those areas that are undergoing change.

USE IN HOTEL CHAIN SCENARIO

A change is required to three PDF documents that are stored securely on the website so that customers can log in and download them. The requirements are documented in the requirements management tool. Testers obtain the requirements from the tool and begin to devise test conditions and test cases. A subsequent change means that further changes are made to the requirements. The testers should be made aware of the changes so that they can provide input to the impact analysis. Traceability within a requirements management tool will also highlight the test conditions affected by the changed requirement. The testers can review the change in requirements and then consider what changes need to be made to the test conditions and test cases.

Configuration management tools and continuous integration tools Configuration management tools are designed primarily for managing the versions of different software (and hardware) components that comprise a complete build of the system; and various complete builds of systems that exist for various software platforms over a period of time.

Continuous integration tools have been developed more recently and can be used in conjunction with configuration management tools to ensure that the correct versions of different programs are integrated into the Daily Build that is deployed into the test environment. This is particularly advantageous for Agile developments where it is important to produce builds automatically and quickly.

A build is a development activity where a complete system is compiled and linked (typically daily) so that a consistent system is available at any time including all the latest changes.

USE IN HOTEL CHAIN SCENARIO

Within the hotel booking system, there will be many versions of sub-systems due to the date at which the version was included in a build, or the operating system on which the version works and so on. Each version of a sub-system will have a unique version number and will comprise many different components (e.g. web services, program files, data files, etc.).

The configuration management tool maps the version number of each sub-system to the build (or release) number of the integrated system. As shown in Table 6.1, Build A (UNIX) and Build B (Windows Server 2016) might use the same version (v1.02) of the Payments In sub-system, but Release C might use version v1.04.

Table 6.1 Configuration traceability

Build for integrated system	Version of Payments In System	Credit card payment test procedure ID	Electronic payment test procedure ID
Build A	v1.02	TP123a	TP201
Build B	v1.02	TP123b	TP201
Build C	v1.04	TP123b	TP201

The same principle applies to testware with a different version number for a test procedure being used, depending on the version number of the build. For instance, test procedure TP123a might be used for Build A and TP123b might be used for Build B – even though both have the same purpose and even expected results. However, another test procedure, TP201, may be applicable to all builds.

A continuous integration tool will support the Agile development methods being used for the mobile app so that deployments into the test environment can be done automatically and quickly.

The amount of benefit to be obtained from using configuration management tools and continuous integration tools is largely dependent on:

- the complexity of the system architecture;
- the number and frequency of builds of the integrated system;
- how much choice (options) is available to customers (whether internal or external).

For example, a software house selling different versions of a product to many customers who run on a variety of operating systems is likely to find configuration management tools more useful than an internal development department working on a single operating system for a single customer. However, an internal development department using an Agile approach will find continuous integration tools almost essential for managing frequent deployments (Daily Builds) into the test environment.

The use of configuration management tools allows traceability between testware and builds of an integrated system and versions of sub-systems and modules. Traceability is useful for:

- identifying the correct version of test procedures to be used;
- determining which test procedures and other testware can be reused or need to be updated/maintained;
- assisting the debugging process so that a failure found when running a test procedure can be traced back to the appropriate version of a sub-system.

CHECK OF UNDERSTANDING

1. What is traceability?
2. Which tool is likely to be most closely integrated with a requirements management tool?
3. Which tool would you use to identify the version of the software component being tested?
4. Which tool would you use to produce a Daily Build?

Tool support for static testing

Tools that support reviews Tools that support reviews (also known as review tools or review process support tools in previous versions of the syllabus) provide a framework for reviews or inspections. This can include:

- maintaining information about the review process, such as rules and checklists;
- the ability to record, communicate and retain review comments and defects;
- the ability to amend and reissue the deliverable under review while retaining a history or log of the changes made;
- traceability functions to enable changes to deliverables under review to highlight other deliverables that may be affected by the change;
- the use of web technology to provide access from any geographical location to this information.

Review tools can interface with configuration management tools to control the version numbers of a document under review.

If reviews and inspections are already performed effectively, then a review tool can be implemented fairly quickly and relatively cheaply. However, if such a tool is used as a means for imposing the use of reviews, then the training and implementation costs will be fairly high (as is the case for implementing a review process without such tools). These tools support the review process, but management buy-in to reviews is necessary if benefits from them are to be obtained in the long run.

Review tools tend to be more beneficial for peer (or technical) reviews and inspections rather than walkthroughs and informal reviews.

USE IN HOTEL CHAIN SCENARIO

The hotel company could use a review tool to perform a review of a system specification written in the UK, so that offshore developers can be involved in the review process. In turn, the review of program code, written offshore, could also be performed using such a tool. This means that both the UK and offshore staff could be involved in both reviews, with no excuses for the right people not being available to attend.

Static analysis tools Static analysis tools (also known as static code analysers) analyse code before it is executed in order to identify defects as early as possible. Therefore, they are used mainly by developers prior to unit testing. A static analysis tool generates lots of error and warning messages about the code. Training may be required in order to interpret these messages and it may also be necessary to configure the tool to filter out particular types of warning messages that are not relevant. The use of static analysis tools on existing or amended code is likely to result in lots of messages concerning programming standards. A way of dealing with this situation should be considered during the selection and implementation process. For instance, it may be agreed that small changes to existing code should not use the static analysis tool, whereas medium to large changes to existing code should use the static analysis tool. A rewrite should be considered if the existing code is significantly non-compliant.

Static analysis tools can find defects that are hard to find during dynamic testing. They can also be used to enforce programming standards (including secure coding), to improve the understanding of the code and to calculate complexity and other metrics (see **Chapter 3**).

Some static analysis tools are integrated with dynamic analysis tools and coverage measurement tools. They are usually language specific, so to test code written in C++ you need to have a version of a static analysis tool that is specific to C++.

Other static analysis tools come as part of programming languages or only work with particular development platforms. Note that debugging tools and compilers provide some basic functions of a static analysis tool, but they are generally not considered to be test tools and are excluded from the ISTQB syllabus.

The types of defect that can be found using a static analysis tool can include:

- Syntax errors (e.g. spelling or missing punctuation).
- Variance from programming standards (e.g. too difficult to maintain).
- Invalid code structures (missing ENDIF statements).
- The structure of the code means that some modules or sections of code may not be executed. Such unreachable code or invalid code dependencies may point to errors in code structure.
- Portability (e.g. code compiles on Windows but not on UNIX).
- Security vulnerabilities.
- Inconsistent interfaces between components (e.g. XML messages produced by component A are not of the correct format to be read by component B).
- References to variables that have a null value or variables declared but never used.

USE IN HOTEL CHAIN SCENARIO

Static analysis tools may be considered worthwhile for code being developed by offshore development teams who are not familiar with in-house coding standards. Such tools may also be considered beneficial for high-risk functions such as BACS and other external interfaces.

CHECK OF UNDERSTANDING

1. Which of the tools used for static testing is/are most likely to be used by developers rather than testers?
2. In which part of the test process are static analysis tools likely to be most useful?

Tool support for test design and implementation

Test design tools Test design tools are used to support the generation and creation of test cases. In order for the tool to generate test cases, a test basis needs to be input and maintained. Therefore, many test design tools are integrated with other tools that already contain details of the test basis such as:

- requirements management tools;
- static analysis tools;
- test management tools.

The level of automation can vary and depends on the characteristics of the tool itself and the way in which the test basis is recorded in the tool. For example, some tools allow specifications or requirements to be specified in a formal language. This can allow test cases with inputs and expected results to be generated. Other test design tools allow a GUI model of the test basis to be created and then allow tests to be generated from this model.

Some tools (sometimes known as test frames) merely generate a partly filled template from the requirement specification held in narrative form. The tester will then need to add to the template and copy and edit as necessary to create the test cases required.

Tests designed from database, object or state models held in modelling tools can be used to verify that the model has been built correctly and can be used to derive some test cases. Tests derived can be very thorough and give high levels of coverage in certain areas.

Some static analysis tools integrate with tools that generate test cases from an analysis of the code. These can include test input values and expected results.

A test oracle [Not in Syllabus] is a type of test design tool that automatically generates expected results. However, these are rarely available because they perform the same function as the software under test. Test oracles tend to be most useful for:

- replacement systems;
- migrations;
- regression testing.

USE IN HOTEL CHAIN SCENARIO

A test oracle could be built using a spreadsheet to support the testing of customers' bills. The tester can then input details for calculating bills, such as the total bill based on various transaction types, refunds, VAT and so on. The spreadsheet could then calculate the total bill amount and this should match the bill calculated by the system under test.

However, test design tools should be only part of the approach to test design. They need to be supplemented by other test cases designed with the use of other techniques and the application of risk.

Test design tools could be used by the test organisation in the scenario but the overhead to input the necessary data from the test basis may be too great to give any real overall benefit. However, if the test design tool can import requirements or other aspects of the test basis easily, then it may become worthwhile.

Test design tools tend to be more useful for safety-critical and other high-risk software where coverage levels are higher and industry, defence or government standards need to be adhered to. Commercial software applications, like the hotel system, do not usually require such high standards and therefore test design tools are of less benefit in such cases.

Model-based testing tools (also known as modelling tools) Model-based testing tools are used primarily by developers during the analysis and design stages of the development life cycle. The reason modelling tools are included here is because they are very cost-effective at finding defects early in the development life cycle.

Their benefits are similar to those obtained from the use of reviews and inspections, in that modelling tools allow omissions and inconsistencies to be identified and fixed early so that detailed design and programming can begin from a consistent and robust model. This in turn prevents fault multiplication that can occur if developers build from the wrong model.

For instance, a visual model-based testing tool using Unified Modeling Language (UML) can be used by designers to build a model of the software specification. The tool can map business processes to the system architecture model, which, in turn, enables programmers and testers to have a better and common understanding of what programs should do and what testing is required.

Similarly, the use of database, state or object models can help to identify what testing is required and can assist in checking whether tests cover all necessary transactions. Integration with test design tools may also enable modelling tools to support the generation of test cases.

USE IN HOTEL CHAIN SCENARIO

The model-based testing tool could help to identify missing scenarios from letter templates or the need to update letters with new paragraphs. Again, the benefits of a clearly defined, consistent model of the software will assist offshore companies to develop software that meets the requirements of the customer.

The use of modelling tools is particularly useful in complex system architectures such as in this scenario.

Test data preparation tools Test data preparation tools are used by testers and developers to manipulate data so that the environment is in the appropriate state for the test to be run. This can involve making changes to the field values in databases, data

files and so on, and populating files with a spread of data (including depersonalised dates of birth, names and addresses, etc. to support data anonymity).

USE IN HOTEL CHAIN SCENARIO

A set of test data may be created by taking, say, 5 per cent of all records from the live system and scrambling personal details so that data is protected and to ensure that customer letters being tested are not wrongly sent to real customers. Data could be taken from the mainframe system, but it is also very important to retain integrity of data between different systems. Data that is held in other databases would need to remain consistent with records on the mainframe.

The knowledge of the database structure and which fields need to be depersonalised is likely to lie with the development team – so it is important to consider whether to buy a tool or build it within the organisation.

TDD, ATDD and BDD tools As technology has evolved and processes have improved, new test tools have been developed to support TDD and BDD approaches (outlined in syllabus sections 1.4.2 and 2.2.1). Readers should be aware that tools exist to support these processes (although many readers may not have used these processes or such tools because they are configured and used primarily by developers during component testing).

TDD, ATDD and BDD tools can be further integrated with other types of test tools to design, implement and execute test cases. This integration enables code changes to be compiled quickly, accurately and regularly, and the test suite executed automatically.

TDD tools have been built to support the test-driven development process and these tools enable developers to design test cases and test procedures at component level. These tools usually integrate with or provide interfaces to types of test execution tools.

ATDD is an extension of TDD and, consequently, ATDD tools are an extended version of TDD tools.

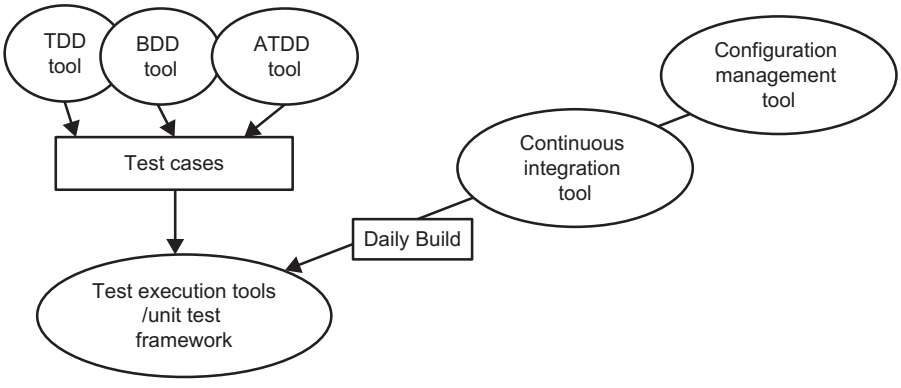
BDD tools are also an extension of TDD tools and provide an interface to allow acceptance testers to define user stories or test cases in their own business language (Domain-Specific Language – DSL). The BDD is then configured by the developer to interpret the user story and produce automatically executable test procedures (scripts).

Developers can configure ATDD and BDD tools to enable users to define test cases in a structured format/template that enables such test cases to be captured by ATDD and BDD tools.

USE IN HOTEL CHAIN SCENARIO

As shown in Figure 6.4 below, a TDD development approach (and a TDD tool) can be used in the Agile workstream for the mobile app. Developers can use the continuous integration tool (and configuration management tool) in conjunction with the TDD tool to design test cases and execute frequent tests of Daily Builds.

Figure 6.4 Testing of daily builds using a set of test tools



ATDD and/or BDD templates can be used by acceptance testers to define test cases in a structured version of their own hotel business language. These structured test cases can then be automated and run by test execution tools/unit test framework, which enables the automated part of acceptance testing to be carried out more efficiently.

CHECK OF UNDERSTANDING

1. What types of inputs can a test design tool use to generate test cases?
2. What is a significant benefit of using model-based testing tools from a testing perspective?
3. In what part of the software development life cycle are test-driven development tools most likely to be used?

Many of the tools discussed in the section above (**Tool support for test design and implementation**) are also used for test execution (or can interface with test execution tools).

Tool support for test execution and logging

Test execution tools cover a wide range from basic test comparators to ATDD tools (see above) that convert acceptance test cases into executable scripts and then report upon whether they have passed or failed.

Test comparators [Not in Syllabus] Test comparators compare the contents of files, databases, XML messages, objects and other electronic data formats. This allows expected results and actual results to be compared. They can also highlight differences and thus provide assistance to developers when localising and debugging code.

They often have functions that allow specified sections of the file, screen or object to be ignored or masked out. This means that a date or time stamp on a screen or field can be masked out because it is expected to be different when a comparison is performed.

Table 6.2 shows an extract from the transaction table in the hotel chain database for data created on 20/10/2018.

Table 6.2 Hotel system extract (20/10/2018)

Transaction ID	Trans_Date	Amount _exc_ VAT	VAT	Customer ID
12345	20/10/2018	359.66	71.93	AG0012
12346	20/10/2018	2312.01	462.40	HJ0007

A regression test was run on 5/11/2018. Table 6.3 shows an extract from the transaction table for this data.

Table 6.3 Hotel system extract (5/11/2018)

Transaction ID	Trans_Date	Amount _exc_ VAT	VAT	Customer ID
12369	5/11/2018	359.66	71.93	AG0012
12370	5/11/2018	2312.01	462.40	HJ0007

The Transaction ID and Trans_Date fields contain different values. But we know why this is the case and we would expect them to be different. Therefore, we can mask out these values. Note that some automated test comparators use test oracles while others provide functions to add on values to take into account known differences (e.g. 15 days later) so that the actual results and expected results can be compared.

Comparators are particularly useful for regression testing since the contents of output or interface files should usually be the same. This is probably the test tool that provides

the single greatest benefit. For instance, manually comparing the contents of a database query containing thousands of rows is time-consuming, error prone and demotivating. The same task can be performed accurately and in a fraction of the time using a comparator. Comparators are usually included in test execution tools, which is why they are no longer specified in the syllabus.

Test execution tools Test execution tools allow test scripts to be run automatically (or at least semi-automatically). A test script (written in a programming language or scripting language) is used to navigate through the system under test and to compare predefined expected outcomes with actual outcomes. The results of the test run are written to a test log. Test scripts can then be amended and reused to run other or additional scenarios through the same system. Some tools offer GUI-based utilities that enable amendments to be made to scripts more easily than by changing code. These utilities may include:

- configuring the script to identify particular GUI objects;
- customising the script to allow it to take specified actions when encountering particular GUI objects or messages;
- parameterising the script to read data from various sources.

Record (or capture playback) tools Record (or capture playback) tools can be used to record a test script and then play it back exactly as it was executed. However, a test script usually fails when played back owing to unexpected results or unrecognised objects. This may sound surprising, but consider entering a new customer record onto a system:

- When the script was recorded, the customer record did not exist. When the script is played back the system correctly recognises that this customer record already exists and produces a different response, thus causing the test script to fail.
- When a test script is played back and actual and expected results are compared a date or time may be displayed. The comparison facility will spot this difference and report a failure.
- Other problems include the inability of test execution tools to recognise some types of GUI control or object. This might be able to be resolved by coding or reconfiguring the object characteristics (but this can be quite complicated and should be left to experts in the tool).

Also note that expected results are not necessarily captured when recording user actions and therefore may not be compared during playback.

The recording of tests can be useful during exploratory testing for reproducing a defect or for documenting how to execute a test. In addition, such tools can be used to capture user actions so that the navigation through a system can be recorded. In both cases, the script can then be made more robust by a technical expert so that it handles valid system behaviours depending on the inputs and the state of the system under test.

Data-driven testing Robust test scripts that deal with various inputs can be converted into data-driven tests. This is where hard-coded inputs in the test script

are replaced with variables that point to data in a data-table. Data-tables are usually spreadsheets with one test case per row, with each row containing test inputs and expected results. The test script reads the appropriate data value from the data-table and inserts it at the appropriate point in the script (e.g. the value in the Customer Name column is inserted into the Customer Name field on the input screen).

Keyword-driven testing A further enhancement to data-driven testing is the use of keyword-driven (or action word) testing. Keywords are included as extra columns in the data-table. The script reads the keyword and takes the appropriate actions and subsequent path through the system under test. Conditional programming constructs such as IF ELSE statements or SELECT CASE statements are required in the test script for keyword-driven testing.

Technical skills Programming skills and programming standards are required to use the tool effectively. It may be that these can be provided by a small team of technical experts within the test organisation or from an external company. In data-driven, and particularly keyword-driven, approaches, the bulk of the work can be done by manual testers, with no knowledge of the scripting language, defining their test cases and test data and then running their tests and raising defects as required. However, this relies on robust and well-written test scripts that are easy to maintain. This takes much time and effort before any sort of payback is achieved.

Maintenance It is essential that time (and consequently budget) is allowed for test scripts to be maintained. Any change to a system can mean that the test scripts need to be updated. Therefore, the introduction of a new type of object or control could result in a mismatch being found between the previous object type and the new one. The relevant level of technical skills and knowledge is also required to do this.

Effective and efficient use The efficiency and effectiveness benefits that come from the use of a test execution tool take a long time to come to fruition. First, the selection and implementation process needs to be planned and conducted effectively (a generic process for selecting and implementing any type of test tool is detailed later in this chapter). However, there are certain issues that are particularly relevant to test execution tools and these are described below.

The long-term benefits of test execution tools include:

- cost savings as a result of the time saved by running automated tests rather than manual tests;
- accuracy benefits from avoiding manual errors in execution and comparison;
- the ability and flexibility to use skilled testers on more useful and interesting tasks (than running repetitive manual tests);
- the speed with which the results of the regression pack can be obtained.

Note that benefits come primarily from running the same or very similar tests a number of times on a stable platform. Therefore, they are generally most useful for regression testing.

USE IN HOTEL CHAIN SCENARIO

Let us assume that a test execution tool is already used for regression testing. Existing automated test scripts could be analysed to identify which ones can be reused and to identify gaps in the coverage for the new enhancement. These gaps could be filled by running cases manually or by writing new automated test scripts. Rather than starting from scratch, it **may be possible** to produce additional automated scripts by reusing some code or modules already used by existing scripts, or by using parameterisation and customisation utilities. In this enhancement, the automated scripts used to test the unchanged documents **could** be run without having to be amended.

The automated scripts to produce the amended documents would need to be analysed and updated as required. The navigation part of the script would be largely unchanged but the comparison between actual and expected results would probably be performed manually the first time round. Once the test has passed manually, the comparison could be added to the script for reuse in the future.

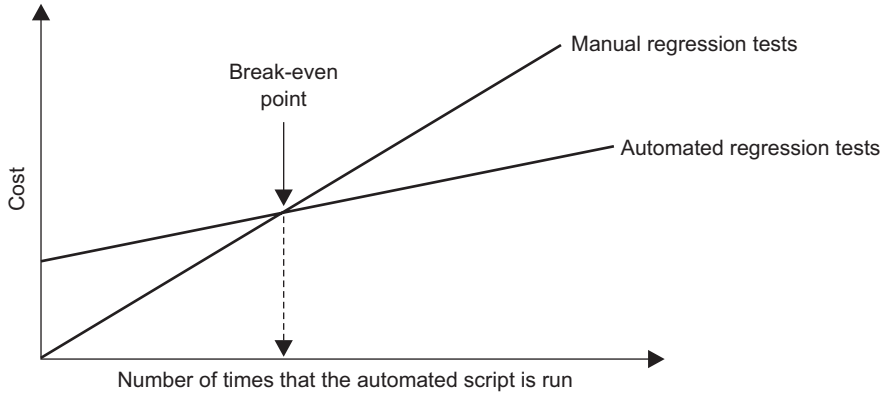
Automated scripts for new documents could be added to the regression pack after this release is complete.

The graph in Figure 6.5 shows how the benefits of using test execution tools take some time to pay back. Note how in the early stages the cost of using automated regression testing is greater than the cost of manual regression testing. This is due to the initial investment, implementation, training, initial development of automated scripts and so on. However, the cost each additional time the test is run is less for automated regression testing than it is for manual regression testing. Therefore, the lines on the graph converge and at a point in time (known as the break-even point) the lines cross. This is the point at which the total cost to date for automated testing is less than the total cost to date for manual regression testing.

This is clearly a simplistic view, but it demonstrates how an initial investment in test execution tools can be of financial benefit in the medium to long term. There are other less tangible benefits as well. However, to get this financial benefit you will need to be sure that there is a requirement to run the same (or very similar) regression tests on many occasions.

Test harnesses and unit test frameworks A test harness is a test environment comprised of stubs and drivers needed to execute a test. It is used primarily by developers to simulate a small section of the environment in which the software will operate. Test harnesses are usually written in-house by developers to perform component or integration testing for a specific purpose. Test harnesses often use 'mock objects' known as 'stubs' (which stub out the need to have other components or systems by returning predefined values) and 'drivers' (which replace the calling component or system and drive transactions, messages and commands through the software under test).

Figure 6.5 Test execution tools payback model



Test harnesses can be used to test various systems or objects ranging from a middleware system (as in Figure 6.6) to a single or small group of components. They are frequently used in Agile development so that existing tests can be rerun as regression tests to establish whether existing functionality is adversely impacted by the changes made.

A unit test framework is generally more robust and reusable than a standalone test harness and is typically able to support multiple test harnesses for related purposes. It may also provide additional support for the developer, such as debugging capabilities.

USE IN HOTEL CHAIN SCENARIO

Bookings are entered via the web or GUI front-ends and are loaded onto the mainframe. An overnight batch runs on the mainframe and generates XML messages that are then processed by the middleware system, which makes a further call to the mainframe to read other data. The middleware system then generates further XML messages, which are processed by other systems, resulting in the production of letters to customers.

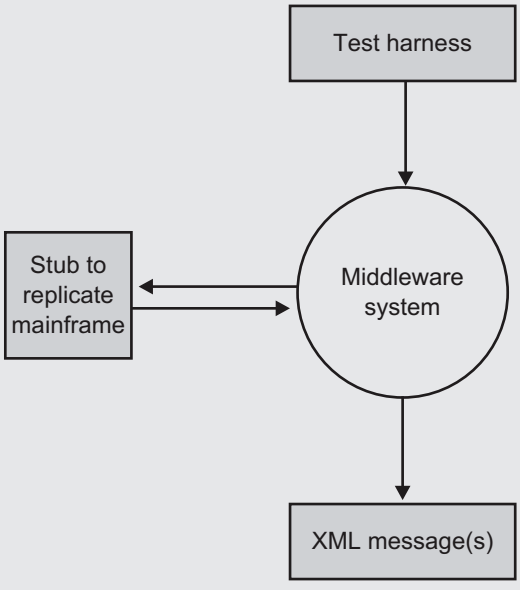
There are several benefits that can be obtained from using a test harness that generates the XML messages produced by the mainframe:

- It would take a lot of time and effort to design and execute test cases on the mainframe system and run the batch.
- It would be costly to build a full environment.
- The mainframe code needed to generate the XML messages may not yet be available.

- A smaller environment is easier to control and manage. It enables developers (or testers) to perform component and integration testing more quickly because it is easier to localise defects. This allows a quicker turnaround time for fixing defects.

The diagram in Figure 6.6 shows that a test harness has been built using a spreadsheet and macros (the driver) to generate XML messages and send them to the middleware. A stub is used to simulate the calls made by the middleware to the mainframe. The contents of the XML messages produced by the middleware can then be compared with the expected results. This could be enhanced into a more robust and reusable unit test framework that can support additional test harnesses for multiple XML messages.

Figure 6.6 Test harness for middleware



There are similarities with the principle behind data-driven testing using test execution tools because the harness allows many different test cases to be designed and run without the time-consuming process of keying them manually. This raises the question of how much benefit can be obtained from using a test execution tool when a test harness can be used instead. As usual, it depends on the circumstances, the risk, the purpose and the level of testing being performed.

Coverage tools Coverage tools measure the percentage of the code structure covered across white-box measurement techniques such as statement coverage and branch or decision coverage. In addition, they can also be used to measure coverage of modules and function calls. Coverage tools are often integrated with static and dynamic analysis tools and are primarily used by developers.

Coverage tools can measure code written in several programming languages, but not all tools can measure code written in all languages. They are useful for reporting coverage measurement and can therefore be used to assess test completion criteria and/or exit criteria.

Coverage measurement of requirements and test cases/scripts run can usually be obtained from test management tools. This function is unlikely to be provided by coverage tools.

Coverage measurement can be carried out using intrusive or non-intrusive methods. Non-intrusive methods typically involve reviewing code and running code. Intrusive methods, such as 'instrumenting the code' involve adding extra statements into the code. The code is then executed and the extra statements write back to a log in order to identify which statements and branches have been executed.

Instrumentation code can then be removed before it goes into production.

Intrusive methods can affect the accuracy of a test because, for example, slightly more processing will be required to cope with the additional code. This is known as the probe effect and testers need to be aware of its consequences and try to keep its impact to a minimum.

USE IN HOTEL CHAIN SCENARIO

Coverage tools are generally used on high-risk and safety-critical systems and therefore would probably not be used in the **Hotel Chain Scenario**. However, as an example, assume that the exit criteria for a test phase include the criteria shown in Table 6.4.

Tool support for performance measurement and dynamic analysis

Dynamic analysis tools Dynamic analysis tools are used to detect the type of defects that are difficult to find during static testing. They are typically used by developers, during component testing and component integration testing, to:

- report on the state of software during its execution;
- monitor the allocation, use and deallocation of memory;
- identify memory leaks;

Table 6.4 Exit criteria

Function	Module risk level	Branch coverage	Statement coverage
BACS	High	100%	100%
Email message to selected customers	Medium	Not specified	100%
Look-up error message/screen navigation	Low	50%	75%

In this case, coverage tools would be the most appropriate method of assessing whether the exit criteria have been met.

- detect time dependencies;
- identify unassigned pointers;
- check pointer arithmetic.

They are generally used for safety-critical and other high-risk software where reliability is critical.

Dynamic analysis tools are often integrated with static analysis and coverage tools. For example, a developer may want to perform static analysis on code to localise defects so that they can be removed before component test execution. The integrated tool may allow:

- the code to be analysed statically;
- the code to be instrumented;
- the code to be executed (dynamically).

Dynamic analysis tools are usually language specific, so to test code written in C++ you need to have a version of a dynamic analysis tool that is specific to C++.

The tool could then:

- report static defects;
- report dynamic defects;
- provide coverage measurement figures;
- report on the code being (dynamically) executed at various instrumentation points.

USE IN HOTEL CHAIN SCENARIO

The hotel chain would probably not use dynamic analysis tools as the benefits for a normal commercial software system (such as this) are relatively small compared with the investment and ongoing costs of dynamic testing tools. However, if static analysis and coverage tools are used, then the additional cost of using dynamic analysis tools may be reduced because they usually come in the same package. Another contributory factor in the decision is that the work done during static analysis and coverage measurement may mean that little additional effort is required to run dynamic tests.

Performance testing tools Performance testing is very difficult to do accurately and in a repeatable way without using test tools. Therefore, performance testing tools have been developed to carry out both load testing and stress testing.

Load testing reports on the performance of a system under test, under various loads and usage patterns. A load generator (which is a type of test driver) can be used to simulate the load and required usage pattern by creating virtual users that execute predefined scripts across one or more test machines. Alternatively, response times or transaction times can be measured under various levels of usage by running automated repetitive test scripts via the user interface of the system under test. In both cases output will be written to a log. Reports or graphs can be generated from the contents of the log to monitor the level of performance.

Performance testing tools can also be used for stress testing. In this case, the load on the system under test is increased gradually (ramped up) in order to identify the usage pattern or load at which the system under test fails. For example, if an air traffic control system supports 200 concurrent aircraft in the defined air space, the entry of the 201st or 205th aircraft should not cause the whole system to fail.

Performance testing tools can be used against whole systems, but they can also be used during system integration test to test an integrated group of systems, one or more servers, one or more databases or a whole environment.

If the risk analysis finds that the likelihood of performance degradation is low, then it is likely that no performance testing will be carried out. For instance, a small enhancement to an existing mainframe system does not necessarily require any formal performance testing. Normal manual testing may be considered sufficient (during which poor performance might be noticed).

There are similarities between performance testing tools and test execution tools in that they both use test scripts and data-driven testing. They can both be left to run unattended overnight and both need a heavy upfront investment, which will take some period of time to pay back.

Performance testing tools can find defects such as:

- general performance problems;
- performance bottlenecks;
- memory leakage (e.g. if the system is left running under heavy load for some time);
- record-locking problems;
- concurrency problems;
- excess usage of system resources;
- exhaustion of disk space.

The cost of some performance tools is high, and the implementation and training costs are also high. In addition, finding experts in performance testing is not that easy. Therefore, it is worth considering using specialist consultancies to come in and carry out performance testing using such tools.

USE IN HOTEL CHAIN SCENARIO

The likelihood of poor website performance and the cost of lost business and reputation are likely to be sufficient to justify the use of performance testing to mitigate this risk. Performance testing can range from using a relatively cheap tool to indicate whether performance has improved or deteriorated as a result of the enhancement, to an extensive assessment of response times under normal or maximum predicted usage and identification of the usage pattern that will cause the system to fail.

It is likely that performance test tools will have been used when the website was first developed. Therefore, it may be easy to reuse existing tools to do a regression test of performance. If performance tools were not used when the website was developed it is unlikely to be worthwhile buying and implementing expensive performance testing tools.

An alternative option would be to use tools that already exist on servers or in the test environment to monitor performance. It may also be worth considering using external consultants.

Monitoring tools Monitoring tools are used to check whether whole systems or specific system resources are available and whether their performance is acceptable. Such tools are mainly used in live rather than test environments and are therefore not really testing tools. They tend to be used for monitoring ecommerce, ebusiness or websites, as such systems are more likely to be affected by factors external to the organisation and the consequences can be severe in terms of business lost and bad publicity. Generally, if a website is not available, customers will not report it but will go elsewhere. For example, it was reported in 2003 that a well-known online retailer would lose sales of $660,000 per hour if it were offline during the US trading day. The advent of websites that continually check the status of popular websites means that such information is easily available to the general public.

The use of monitoring tools is generally less important for internal systems because failure is more likely to be noticed only within the organisation and contingency plans may also exist. However, the availability of monitoring tools on mainframes, servers and other forms of hardware means that it is relatively easy to monitor the majority of an organisation's infrastructure.

USE IN HOTEL CHAIN SCENARIO

A monitoring tool may be beneficial to monitor the website. A monitoring tool may also exist as part of the mainframe system. However, it is less likely that monitoring tools will be used for the GUI front-end that is used by internal staff.

CHECK OF UNDERSTANDING

1. Describe two types of defect that can typically be found using dynamic analysis tools.
2. Describe two drawbacks associated with performance testing tools.
3. Which of the tools that provide support for performance and monitoring is most likely to be used by developers?

Tool support for specialised testing needs

In addition to tools that support the general test process, there are many other tools that support more specific testing issues.

Data quality assessment tools Data quality assessment tools allow files and databases to be compared against a format that is specified in advance. They are typically used on large-scale, data-intensive projects such as:

- conversion of data used on one system into a format suitable for another system;
- migration of data from one system to another;
- loading of data into a data warehouse.

Data quality assessment tools are not specifically designed for testing purposes. They are used primarily for the migration of production data, but typically the development and testing of a migration project will also use these tools.

USE IN HOTEL CHAIN SCENARIO

Suppose that the hotel chain buys a smaller group of hotels, 'Small Hotel Group'.

It could use a data quality assessment tool during the development and testing of an enhancement to its existing systems to include the additional hotels.

The data quality assessment tools could be configured to establish whether the customer data being migrated meets particular quality requirements. These requirements may include:

- valid postcodes;
- valid title for gender;
- numeric values in financial fields;
- numeric values in date fields.

The tool could also be used to reconcile file record counts with data held in header and footer records to confirm that the number of records in the file equals the number of records loaded into the database.

Data conversion and migration tools Data conversion tools are used to map data from the data structures in the source system into the data structures required for the receiving system.

Security testing tools Security testing tools are used to test the functions that detect security threats and to evaluate the security characteristics of software. The security testing tool is typically used to assess the ability of the software under test to:

- handle computer viruses;
- protect data confidentiality and data integrity;
- prevent unauthorised access;
- carry out authentication checks of users;
- remain available under a denial of service (DOS) attack;
- check non-repudiation attributes of digital signatures.

Security testing tools are mainly used to test ecommerce, ebusiness and websites. For example, a third-party security application such as a firewall may be integrated into a web-based application.

USE IN HOTEL CHAIN SCENARIO

Security testing tools could be used to test that the firewall and other security applications built into the hotel chain's systems can:

- resist a DOS attack;
- prevent unauthorised access to data held within the database;
- prevent unauthorised access to encrypted XML messages containing bank account details.

This work could be carried out by a third-party consultancy that specialises in penetration testing and related services.

The skills required to develop and use security testing tools are very specialised and such tools are generally developed and used on a particular technology platform for a particular purpose. Therefore, it may be worth considering the use of specialist consultancies to perform such testing. For example, specialist consultancies are often engaged to carry out penetration testing. This type of testing is to establish whether malicious attackers can penetrate the organisation's firewall and hack into its systems.

Security testing tools need to be constantly updated because there are problems solved and new vulnerabilities discovered all the time – consider the number of Windows security releases to see the scale of security problems.

Usability testing tools (including accessibility and localisation test tools) Usability test tools typically record the mouse clicks made by remote usability testers when carrying out a specified task. Some tools also enable other data to be captured such as screenshots, written comments and voice recordings of verbal comments. This recorded data is generally stored in a database so that it can then be analysed easily by staff at the organisation commissioning the testing.

Note that the usability testing tool market is changing very quickly, and new types of usability tools may appear over the next few years. Recent developments include:

- accessibility test tools – which are an extension to usability test tools. Accessibility testing is defined as testing to determine the ease by which users with disabilities can use a component or system;
- localisation test tools – which have been developed to support the testing of local language versions of widely available global software products.

Portability testing tools Portability is defined as the ease with which the software product can be transferred from one hardware or software environment to another. Portability test tools are generally used by Portability testing specialists.

USE IN HOTEL CHAIN SCENARIO

The changes to the website to improve usability could be tested by a specialist usability testing company who employ, say 50, remote users. The remote users would be given a high-level requirement that would exercise the website changes such as:

- Go to a specified test URL and book three rooms from 3 August to 5 August and two rooms from 7 August. Pay by credit card XYZ.

The mouse clicks, other inputs and comments recorded by the 50 remote users in carrying out this task would be stored in a database and an analysis report produced by the specialist usability testing company for the hotel chain. This analysis could highlight poor areas of usability in the test website, which could be improved before being deployed to the live website.

Other tools that are not designed specifically for testers or developers can also be used to support one or more test activities. These include:

- spreadsheets;
- word processors;
- email;
- back-up and restore utilities;
- SQL and other database query tools;
- project planning tools;
- debugging tools (although these are more likely to be used by developers than testers).

For example, in the absence of test management or defect management tools, defects could be recorded on word processors and could be tracked and maintained on spreadsheets. Tests passed or failed could also be recorded on spreadsheets.

USE IN HOTEL CHAIN SCENARIO

Other software tools could also be used:

- A spreadsheet could be used for producing decision tables or working out all the different test scenarios required. It could also be used to manipulate test management information so that it can be presented in the format required in weekly or daily test progress reports.
- Word processors could be used for writing test strategies, test plans, weekly reports and other test deliverables.

- Email could be used for communicating with developers about defects and for distributing test reports and other deliverables.
- Back-up and restore utilities could be used to restore a consistent set of data into the test environment for regression testing.
- SQL could be used for analysing the data held in databases in order to obtain actual or expected results.
- Project planning tools could be used to estimate resources and timescales, and monitor progress.
- Debugging tools can be used by developers to localise and fix defects.

CHECK OF UNDERSTANDING

Name four tools that are not specifically designed for testers. Give an example of how each of them could be of use to a tester.

Summary of test tools

Table 6.5 summarises the types of test tools discussed above. It includes the definition given in the ISTQB Glossary of Testing Terms v3.2 and gives a guide to:

- the main ISTQB syllabus classification;
- the activity in the test process for which the tool is usually most useful;
- the most likely users of the tool.

INTRODUCING A TOOL INTO AN ORGANISATION

There are many stages in the process that should be considered before implementing a test tool.

Analyse the problem/opportunity

An assessment should be made of the maturity of the test process used within the organisation. If the organisation's test processes are immature and ineffective then the most that the tool can do is to make the repetition of these processes quicker and more accurate – quick and accurate ineffective processes are still ineffective.

It is therefore important to identify the strengths, weaknesses and opportunities that exist within the test organisation before introducing test tools. Tools should only be implemented that will either support an established test process or support required improvements to an immature test process. It may be beneficial to carry out a TPI (Test Process

Table 6.5 Types of test tool

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Test management and application life cycle management (ALM)	Tool support for management of testing and testware	All activities	A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting.	Testers
Defect management	Tool support for management of testing and testware	Primarily test execution – because that is where most defects are found	A tool that facilitates the recording and status tracking of defects.	Various, but particularly testers
Requirements management	Tool support for management of testing and testware	Test analysis – to determine what requirements are in scope for testing	A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to predefined requirements rules.	Various, but particularly business analysts

(Continued)

Table 6.5 (Continued)

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Configuration management	Tool support for management of testing and testware	Test implementation – deploying the appropriate version of software is key to building the test environment	A tool that provides support for the identification and control of configuration items, their status over changes and versions, and the release of baselines consisting of configuration items. A configuration item is an aggregation of work products that is designated for configuration management and treated as a single entity in the configuration management process.	Various – including release managers, developers
Continuous integration	Tool support for management of testing and testware	Test implementation deploying the appropriate version of software is key to building the test environment	Continuous integration tools provide regular and frequent builds. A build is defined as a development activity where a complete system is compiled and linked (typically daily) so that a consistent system is available at any time including all the latest changes.	Developers
Tools that support reviews	Tool support for static testing	Most activities – reviews can take place in most parts of the test process – but is not normally part of test execution	A tool that provides support to the review process. Typical features include review planning and tracking support, communication support, collaborative reviews and a repository for collecting and reporting of metrics.	Various

(Continued)

Table 6.5 (Continued)

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Static analysis	Tool support for static testing	Static analysis is done without executing the code, so is part of test implementation	A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics and data flow anomalies.	Developers
Test design	Tool support for test design and implementation	Test design	A tool that supports the test design activity by generating test inputs from a specification held in a CASE (Computer Aided Software Engineering) tool repository, e.g. requirements management tool, from specified test conditions held in a tool itself, or from code.	Testers
Model-based testing	Tool support for test design and implementation	Test analysis, test design and test implementation	A tool that supports the creation, amendment and verification of models of the software or system (previously called modelling tools).	Developers
Test data preparation	Tool support for test design and implementation	Test implementation – preparing test data is a key part of test implementation	A type of test tool that enables data to be selected from existing databases or created, generated, manipulated and edited for use in testing.	Various

(Continued)

Table 6.5 (Continued)

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Test-driven development (TDD)	Tool support for test design and implementation	Mainly test design, but some tools can also be used in test implementation and test execution	Test-driven development is defined as a way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.	Developers
Acceptance test-driven development (ATDD) and behaviour-driven development (BDD)	Tool support for test design and implementation	Mainly test design, but some tools can also be used in test implementation and test execution	ATDD and BDD tools are an enhancement to TDD tools but are configured so that they can be used by testers.	Testers
Test execution (e.g. to run regression tests)	Tool support for test execution and logging	Test execution	A test tool that executes tests against a designated test item and evaluates the outcomes against expected results and postconditions.	Testers
Test harnesses	Tool support for test execution and logging	Test execution	A test environment composed of stubs and drivers needed to conduct a test.	Developers
Unit test framework	Tool support for test execution and logging	Test execution	A unit test framework is generally more robust and reusable than a test harness and is typically able to support multiple test harnesses for related purposes. It may also provide additional support for the developer such as debugging capabilities.	Developers

(Continued)

Table 6.5 (Continued)

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Coverage (e.g. requirements coverage, code coverage)	Tool support for test execution and logging	Test execution	A tool that provides objective measures of what structural elements, e.g. statements, branches, have been exercised by a test suite.	Developers
Dynamic analysis	Tool support for performance measurement and dynamic analysis	Test execution	A tool that provides run-time information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic and to monitor the allocation, use and deallocation of memory and to flag memory leaks.	Developers
Performance testing	Tool support for performance measurement and dynamic analysis	Test execution	A test tool that generates load for a designated test item and that measures and records its performance during test execution.	Performance testing specialists
Monitoring	Tool support for performance measurement and dynamic analysis	Test execution	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.	Various
Data quality assessment/data conversion and migration	Tool support for specialised testing needs	Test implementation and test execution – mainly for data migration projects	Data quality assessment tools allow files and databases to be compared against a format that is specified in advance. Data conversion tools are used to map data from the data structures in the source system into the data structures required for the receiving system. Both are used primarily for the migration of production data, but typically the development and testing of a migration project will also use these tools.	Various

(Continued)

Table 6.5 (Continued)

Test tool type	ISTQB syllabus classification	Activity in the test process where it is usually most helpful	ISTQB Glossary of Testing Terms and related definitions	Most likely users
Security testing	Tool support for specialised testing needs	Mainly test execution – but also used in other stages of the test process	A tool that provides support for testing security characteristics and vulnerabilities.	Security testing specialists
Usability testing	Tool support for specialised testing needs	Mainly test execution – but also used in other stages of the test process	Usability testing is defined as testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.	Usability testing specialists
Accessibility testing	Tool support for specialised testing needs	Mainly test execution – but also used in other stages of the test process	Accessibility testing is defined as testing to determine the ease by which users with disabilities can use a component or system.	Accessibility testing specialists
Localisation testing	Tool support for specialised testing needs	Mainly test execution – but also used in other stages of the test process	Localisation testing is not clearly defined, but in general terms these tools focus on supporting the testing of local language versions of widely available global software products.	Localisation testing specialists
Portability testing (e.g. testing software across multiple supported platforms)	Tool support for specialised testing needs	Mainly test execution – but also used in other stages of the test process	Portability is defined as the ease with which the software product can be transferred from one hardware or software environment to another.	Portability testing specialists

Improvement) or CMMI (Capability Maturity Model Integration) assessment to establish the maturity of the organisation before considering the implementation of any test tool.

Generate alternative solutions

It may be more appropriate and cost-effective to do something different. In some organisations, performance testing, which may only need to be done from time to time, could be outsourced to a specialist testing consultancy. Training or recruiting better staff could provide more benefits than implementing a test tool and improve the effectiveness of a test process more significantly. In addition, it is more effective to maintain a manual regression pack so that it accurately reflects the high-risk areas than to automate an outdated regression pack (that is no longer relevant) using a test execution tool.

An early investigation of what tools are available is likely to form part of this activity.

Constraints and requirements

A thorough analysis of the constraints and requirements of the tool should be performed. Interested parties should attend workshops and/or be interviewed so that a formal description of the requirements can be produced and approved by the budget holder and other key stakeholders.

A failure to specify accurate requirements (as with a failure to specify accurate requirements for a piece of software) can lead to delays, additional costs and the wrong things being delivered. This could lead to a review tool being implemented that does not allow access across the internet, even though there is a need for staff from many countries to participate in reviews. Any financial or technical constraints (e.g. compatibility with particular operating systems or databases) should also be considered.

It is useful to attach some sort of priority or ranking to each requirement or group of requirements.

Training, coaching and mentoring requirements should also be identified. For example, experienced consultants could be used for a few weeks or months to work on overcoming implementation problems with the tool and to help transfer knowledge to permanent staff. Such consultants could be provided by the vendor or could be from the contract market.

Requirements for the tool vendor should also be considered. These could include the quality of training and support offered by the vendor during and after implementation and the ability to enhance and upgrade the tool in the future. In addition, their financial stability should be considered because the vendor could go bankrupt or sell to another vendor. Therefore, using a small niche vendor may be a higher risk than using an established tool supplier.

If non-commercial tools (such as open source and freeware) are being considered then there are likely to be risks around the lack of training and support available. In addition, the ability or desire of the service support supplier (or open-source provider) to continue to develop and support the tool should be taken into account.

Evaluation and shortlist

The tools available in the marketplace should be evaluated to identify a shortlist of the tools that provide the best fit to the requirements and constraints. This may involve:

- searching the internet;
- attending exhibitions of test tools;
- discussions with tool vendors;
- engaging specialist consultants to identify relevant tools.

It may also be useful for the test organisation to send a copy of its list of requirements and constraints to tool vendors so that:

- the vendor is clear about what the test organisations wants;
- the vendor can respond with clarity about what its own tools can do and what workarounds there are to meet the requirements that the tool cannot provide;
- the test organisation does not waste time dealing with vendors that cannot satisfy its key requirements.

The outcome of this initial evaluation should result in a shortlist of perhaps one, two or three tools that appear to meet the requirements.

Detailed evaluation/proof of concept

A more detailed evaluation (proof of concept) should then be performed against this shortlist. This should be held at the test organisation's premises in the test environment in which the tool will be used. This test environment should use the system under test and other software, operating systems and hardware with which the tool will be used. There are several reasons why there is little benefit from evaluating the tool on something different. For example:

- Test execution tools do not necessarily recognise all object types in the system under test, or they may need to be reconfigured to do so.
- Performance measurement tools may need to be reconfigured to provide meaningful performance information.
- Test management tools may need to have workflow redesigned to support established test processes and may need to be integrated with existing tools used within the test process.
- Static analysis tools may not work on the version of programming languages used.

In some cases, it may be worth considering whether changes can be made to the organisation's test environments and infrastructure, but the costs and risks need to be understood and quantified.

(Note that if there is only one tool in the shortlist then it may be appropriate to combine the proof of concept and the pilot project.)

After each proof of concept the performance of the tool should be assessed in relation to each predefined requirement. Any additional features demonstrated should be considered and noted as potential future requirements.

Once all proofs of concept have been carried out it may be necessary to amend the requirements as a result of what was found during the tool selection process. Any amendments should be agreed with stakeholders. Each tool should then be assessed against the finalised set of requirements.

There are three likely outcomes at this stage:

- None of the tools meets the requirements sufficiently well to make it worthwhile purchasing and implementing them.
- One tool meets the requirement much better than the others and is likely to be worthwhile. In this case select this tool.
- The situation is unclear and more information is needed. In this case a competitive trial or another cycle/iteration of the process may be needed. Perhaps the requirements need to be revised or further questions need to be put to vendors. It may also be time to start negotiations with vendors about costs.

Negotiations with vendor of selected tool

Once a tool has been selected discussions will be held with the vendor to establish and negotiate the amount of money to be paid and the timing of payments. This will include some or all of the following:

- purchase price;
- annual licence fee;
- consultancy costs;
- training costs;
- implementation costs.

Discussions should establish the amount to be paid, first, for a pilot project and, secondly (assuming the pilot project is successful), the price to be paid for a larger scale implementation.

The pilot project

The aims of a pilot project include the following:

- It is important to establish what changes need to be made to the high-level processes and practices currently used within the test organisation. This involves assessing whether the tool's standard workflow, processes and configuration need to be amended to fit with the test process or whether the existing processes need to be changed to obtain the optimum benefits that the tool can provide.

- To determine lower level detail such as templates, naming standards and other guidelines for using the tool. This can take the form of a user guidelines document.
- To establish whether the tool provides value for money. This is done by trying to estimate and quantify the financial and other benefits of using the tool and then comparing this with the fees paid to the vendor and the projected internal costs to the organisation (e.g. lost time that could be used for other things, the cost of hiring contractors etc.).
- A more intangible aim is to learn more about what the tool can and cannot do and how these functions (or workarounds) can be applied within the test organisation to obtain maximum benefit.

The pilot project should report back to the group of stakeholders that determined the requirements of the tool.

If a decision is made to implement the tool on a larger scale then a formal project should be created and managed according to established project management principles.

Key factors in successful implementations of test tools

There are certain factors or characteristics that many successful tool implementation projects have in common:

- Implementing findings from the pilot project such as high-level process changes and using functions or workarounds that can add additional benefits.
- Identifying and subsequently writing user guidelines, based on the findings of the pilot project.
- An incremental approach to rolling out the tool into areas where it is likely to be most useful. For example, this can allow 'quick wins' to be made and good publicity obtained, resulting in a generally positive attitude towards the tool.
- Improving the process to fit with the new tool, or amending the use of the tool to fit with existing processes.
- Ensuring that the appropriate level of training, coaching and mentoring is available. Similarly, there may be a need to recruit permanent or contract resources to ensure that sufficient skills exist at the outset of the tool's use within the organisation.
- Using a database (in whatever format) of problems encountered and lessons learnt to overcome them. This is because new users are likely to make similar mistakes.
- Capturing metrics to monitor the amount of use of the tool. Recording the benefits obtained. This can then be used to support arguments about implementing to other areas within the test organisation.
- Agreeing or obtaining a budget to allow the tool to be implemented appropriately.

Summary of test tool implementation process

The diagram in Figure 6.7 outlines the process for selecting and implementing a test tool in an organisation. This shows that there are several points at which a decision could be

made **not** to introduce a tool. It also demonstrates that the activities during the evaluation and negotiation stages can follow an iterative process until a decision is made.

CHECK OF UNDERSTANDING

1. Why is an understanding of the test organisation's maturity essential before introducing a test tool?
2. What is the purpose of defining requirements for the tool?
3. Why is it important to evaluate the tool vendor as well as the tool itself?
4. What is meant by a proof of concept?
5. What is the purpose of a pilot project?
6. When is it appropriate to combine a proof of concept and pilot project?
7. Name three factors in the successful implementation of tools.

SUMMARY

We have seen that the main benefits of using test tools are generally the same as the benefits of automating a process in any industry. These are: time saved and predictable and consistent results.

However, we have also seen that there can be considerable costs in terms of both time and money associated with obtaining such benefits. The point at which the use of tools becomes economically viable depends on the amount of reuse, which is often difficult to predict.

Other risks include over-optimistic expectations of:

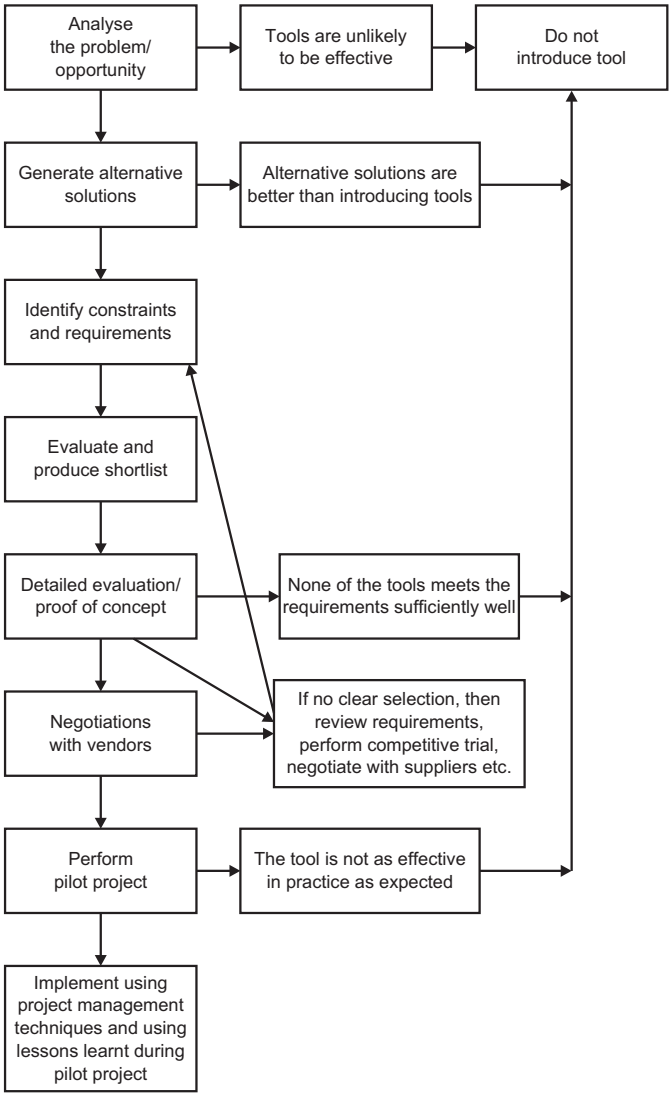
- what the tool can do;
- how easy it is to use;
- the amount of maintenance required.

We have seen that there are many types of test tools and that they provide support to a variety of activities within the test process. We have also seen that tools are used by a variety of staff in the software development process and that some are of greater benefit to developers than testers.

We have looked at the different scripting techniques that can be used with test execution tools. This ranges from the simple record–playback to data-driven and keyword-driven scripts.

We identified a process for selecting and introducing a test tool into an organisation. This involves understanding the interactions between activities within the process and examining the purposes of a proof of concept and of a pilot project. We also examined

Figure 6.7 Test tool implementation process



the problems likely to be encountered when implementing a tool and looked at actions that can be taken in an attempt to overcome or avoid such problems.

We also noted that a decision not to introduce a tool could well be a valid decision at several stages within the process.

Example examination questions with answers

E1. K1 question
A project requires test tools to support both requirements management and usability testing. Which *two* of the following classes of test tool are appropriate to select tools from?

i. Tool support for management of testing and testware.
ii. Tool support for test design and implementation.
iii. Tool support for static testing.
iv. Tool support for specialised testing needs.
v. Tool support for test execution and logging.

a. i and iv.
b. ii and iii.
c. iii and iv.
d. i and v.

E2. K2 question
Which of the following correctly identifies a benefit of test automation?

a. Version control of test assets is no longer required.
b. Greater consistency and repeatability of tests.
c. The tool vendor is always available for help and advice.
d. Regression testing will not be needed.

E3. K1 question
Which of the following is a special consideration for test execution tools?

a. Expertise in scripting languages is required.
b. Test execution tools need to interface with spreadsheets and other tools.
c. Tests cannot be captured by recording the actions of a manual tester.
d. Every tester will need to be trained in the use of the test execution tool.

E4. K2 question
Which of the following is *not* an important principle for test tool selection?

a. Evaluation of the tool against clear requirements and objective criteria.
b. Consideration of pros and cons of various licensing models.
c. Identification of opportunities for an improved test process supported by tools.
d. Identification of changes needed to the tool to ensure that it operates effectively with the existing test process.

E5. K1 question
Which of the following is a typical objective for a pilot project for introduction of a test tool into an organisation?

a. Gaining an understanding of the requirements for a tool.
b. Evaluating how the tool fits with existing processes and practices.
c. Determining what changes will be needed to the tool's functionality.
d. Deciding what benefits the tool might bring.

E6. K1 question
Which of the following is a typical positive success factor for implementation of a test tool within an organisation?

a. Ensuring that the tool is rolled out across the entire organisation simultaneously.
b. Identifying ways to improve the tool once it is implemented.
c. Ensuring that all software development projects make maximum use of the tool.
d. Monitoring tool use and benefits.

Answers to questions in the chapter

SA1. The correct answer is d.
SA2. The correct answer is c.
SA3. The correct answer is a.

Answers to example examination questions

E1. The correct answer is a.
The correct classes of tool are tool support for management of testing and testware (requirements management tool) and tool support for specialised testing needs (usability testing). The only answer that identifies this combination is a.

E2. The correct answer is b.
Option a is incorrect; neglect of version control is listed in section 6.1.2 as a potential risk. Option c is incorrect; vendors going out of business or discontinuing support of a tool are potential risks. Option d is incorrect; regression testing may be easier and quicker to do but will still be required. Option b is specifically listed as a potential benefit of test automation and is the correct answer.

E3. The correct answer is a.
Option a is correct. At least one member of a testing team will need to be able to develop test scripts using a scripting language, so expertise by **some** of the test team is required – but this does not mean **every** tester needs to be able to read/write in any scripting language(s). Option b is incorrect; this is a special consideration for test management tools but not for test execution tools. Option c is incorrect; although capturing manual tests is not the optimum approach to automated test execution, it can be done. Option d

is incorrect; at least one tester will need to be trained to use the tools, but not necessarily the entire team.

E4. The correct answer is d.
Options a, b and c are all key principles listed in the syllabus, section 6.2.1. Option d contradicts option c and is generally considered a dangerous approach that could significantly impair the benefits of using a tool and incur significant costs.

E5. The correct answer is b.
Option a is incorrect; this needs to be done before selecting the tool. Option c is incorrect; changes may be needed to processes and practices, but changes to a tool are expensive and risky. Option d is incorrect because the anticipated benefits would have been determined earlier to support the selection process; at this stage the aim is to determine if the expected benefits can be achieved. The correct option is b, which is listed in section 6.2.2 of the syllabus.

E6. The correct answer is d.
Option a is incorrect; section 6.2.3 of the syllabus suggests that tools should be rolled out incrementally. Option B is incorrect because changes to a tool are potentially expensive and risky; any changes to accommodate the tool should be made to processes and practices. Option c is unlikely to be successful because no tool is likely to be successful in all cases or in all types of project; the syllabus suggests defining guidelines for the use of the tool so that users can best decide where it will add most value. Option d is correct and is listed in the syllabus, section 6.2.3.

7 THE EXAMINATION

THE EXAMINATION

The examination structure

The Certified Tester Foundation Level (CTFL) examination is a one-hour examination made up of 40 multiple choice questions. There are five main aspects to the examination's structure:

- The questions are all equally weighted.
- Questions are set from learning objectives stated in each section.
- The number of questions associated with each section of the syllabus is in proportion to the amount of time allocated to that section of the syllabus, which roughly translates into:
  - Section 1, 8 questions.
  - Section 2, 5 questions.
  - Section 3, 5 questions.
  - Section 4, 11 questions.
  - Section 5, 9 questions.
  - Section 6, 2 questions.

  These proportions are now mandatory.
- The number of questions at each level of understanding will be as follows:
  - K1 20 per cent, that is 8 questions.
  - K2 60 per cent, that is 24 questions.
  - K3 20 per cent, that is 8 questions.

  This is also a mandatory requirement.
- The pass mark is 26 correct answers and there are no penalties for incorrect answers.

The question types

All questions will contain a 'stem', which states the question, and four optional answers. One and only one of the optional answers will be correct. The remainder can be expected to be plausibly incorrect, which means that anyone knowing the correct answer will be unlikely to be drawn to any of the incorrect answers, but anyone unsure of the correct answer will be likely to find one or more alternatives equally plausible.

Questions will be stated as clearly as possible, even emphasising keywords by emboldening where this will add clarity. There should be very few negative questions (e.g. which of the following is **not** true?) and any negative questions included will be worded so that there is no ambiguity. Questions will be set to test your knowledge of the content of the topics covered in the syllabus and not your knowledge of the syllabus itself.

There are no absolute rules for question types as long as they are appropriate to the level of understanding they are testing, but there are some common types of questions that are likely to arise.

As a general rule, K1 questions will be of the straightforward variety shown in the next box.

EXAMPLE OF A K1 QUESTION

(This one is taken from **Chapter 3**.)

What do static analysis tools analyse?

a. Design.
b. Test cases.
c. Requirements.
d. Program code.

(The correct answer is d.)

K2 questions may be of the same type as the K1 example but with a more searching stem. Another form of K2 question is known as the Roman type. This is particularly well suited to questions involving comparisons or testing the candidate's ability to identify correct combinations of information. The example in the next box is a K2 question of the Roman type.

EXAMPLE OF A K2 QUESTION

(This one is taken from **Chapter 3**.)

Which of the following statements are correct for walkthroughs?

i. Often led by the author.
ii. Documented and defined results.
iii. All participants have defined roles.
iv. Used to aid learning.
v. Main purpose is to find defects.

a. i and v are correct.
b. ii and iii are correct.
c. i and iv are correct.
d. iii and iv are correct.

(The correct answer is c.)

K3 questions test the candidate's ability to apply a topic, so the most common form of these is related to test design techniques (though this is not the only topic that can be examined at the K3 level). The next box gives a typical example of a techniques question.

EXAMPLE OF A K3 QUESTION

(This one is taken from **Chapter 4**.)

A system is designed to accept values of examination marks as follows:

Fail	0–39 inclusive
Pass	40–59 inclusive
Merit	60–79 inclusive
Distinction	80–100 inclusive

Which of the following sets of values are all in different equivalence partitions?

a. 25, 40, 60, 75
b. 0, 45, 79, 87
c. 35, 40, 59, 69
d. 25, 39, 60, 81

(The correct answer is b.)

Remember that K1, K2 and K3 do not equate to easy, moderate or hard. The K level identifies the level of understanding being tested, not the difficulty of the question. It is perfectly possible to find K2 questions that are more difficult (in the sense of being more challenging to answer) than a K3 question. It is, however, true that K1 questions will always be the most straightforward and anyone who knows the material in the syllabus should have no difficulty in answering any K1 question. Every question has the same value; any 26 correct answers will guarantee a pass.

Questions in the examination are not labelled by the K level they are testing, but the example questions at the end of each chapter of this book include examples of K1, K2 and K3 questions, and these are labelled by level for your guidance.

The sample examination

A sample examination paper is available from the **ISTQB website**. It is designed to provide guidance on the structure of the paper and the 'rubric' (the rules printed on the front of the paper) of the real examination. The questions in the sample paper are not necessarily typical, though there will be examples of the various types of questions so that candidates are aware of the kinds of questions that can arise. Any topic or type of question in the sample paper can be expected to arise in a real examination at some time. Bear in mind that the sample paper may change from time to time to reflect any changes in the syllabus or to reflect any changes in the way questions are set.

Examination technique

In a relatively short examination there is little time to devote to studying the paper in depth. However, it is wise to pause before beginning to answer questions while you assimilate the contents of the question paper. This brief time of inactivity is also a good opportunity to consciously slow down your heart rate and regulate your breathing; nervousness is natural, but it can harm your performance by making you rush. A few minutes spent consciously calming down will be well repaid. There will still be time enough to answer the questions; a strong candidate can answer 40 questions in less than 45 minutes.

When you do start, go through the whole paper answering those questions that are straightforward and for which you know the answer. When you have done this, you will have a smaller task to complete and you will probably have taken less than a minute for each question that you have already answered, giving you more time to concentrate on those that you will need more time to answer.

Next, turn to those you feel you understand but that will take you a little time to work out the correct answer, and complete as many of those as you can. The questions you are left with now should be those that you are uncertain about. You now know how long you have to answer each of these and you can take a little more time over each of them.

REVISION TECHNIQUES

There are some golden rules for exam revision:

- Do as many example questions as you can so that you become familiar with the types of questions, the way questions are worded and the levels (K1, K2, K3) of questions that are set in the examination.
- Be active in your reading. This usually means taking notes, but this book has been structured to include regular checks of understanding that will provide you with prompts to ensure that you have remembered the key ideas from the section you have just revised. In many cases information you need to remember is already in note form for easy learning.
- One important way to engage with the book is to work through all the examples and exercises. If you convince yourself you can do an exercise, but you do not actually attempt it, you will only discover the weakness in that approach when you are sitting in the examination centre.
- Learning and revision need to be reinforced. There are two related ways to do this:
  - By making structured notes to connect together related ideas. This can be done via lists, but a particularly effective way to make the connections is by using a technique known as mind mapping (there are many free tools on the web or, if you find you really like the technique, you can invest in a more feature-rich product).
  - By returning to a topic that you have revised to check that you have retained the information. This is best done the day after you first revised the topic and again a week after, if possible. If you begin each revision section by returning to the 'Check of understanding' boxes in some or all of the chapters you worked with in previous revision sessions, it will help to ensure that you retain what you are revising.
- Read the syllabus and become familiar with it. Questions are raised directly from the syllabus and often contain wording similar to that used in the syllabus. Familiarity with the syllabus document will more than repay the time you will spend gaining that familiarity.

REVIEW

The layout, structure and style of this book are designed to maximise your learning: by presenting information in a form that is easy to assimilate; by listing things you need to remember; by highlighting key ideas; by providing worked examples; and by providing exercises with solutions. All you need for an intense and effective revision session is in these pages.

The best preparation for any examination is to practise answering as many realistic questions as possible under conditions as close to the real examination as possible. This is one way to use the ISTQB sample paper, or you can construct a sample paper of your own from the questions included in this book. However, the best check of your readiness for tackling the real examination is to attempt the mock exam that is contained in

Appendix A1. All the answers are provided in **Appendix A2** so that you can see how well you did, and a full commentary is provided in **Appendix A3** so that you can identify where you went wrong in any questions.

Good luck with your Foundation Certificate examination.

APPENDICES

A1 MOCK CTFL EXAMINATION

Question 1
Which of the following is most likely to lead to the success of testing?

a. Point out in defect reports why the problems should not have happened.
b. Have a chart of the developers and the number of defects that they have created.
c. Communicate information about defects and failures in a constructive way.
d. Locate testers and developers in separate buildings, to aid internal team building.

Question 2
What is decision table testing?

a. Placing decisions in tables to make them easier to test.
b. A black-box technique to exercise combinations of causes and effects.
c. A table used to show sets of conditions and resulting actions.
d. A white-box test technique to test conditions and their outcomes.

Question 3
Which of the following is true of non-functional testing?

a. Non-functional testing always requires specialist skills.
b. Non-functional testing is technical so should be carried out by developers.
c. Non-functional testing should be carried out at all levels of testing.
d. Due to the nature of non-functional testing, the coverage achieved cannot be measured.

Question 4
Which of the following correctly identifies a key role and associated responsibilities for a *formal* review?

a. Management assigns staff and executes control decisions in the event of inadequate outcomes.
b. Review leader assigns staff and takes overall responsibility for the review.
c. Reviewers, who must be subject matter experts, identify potential defects.
d. A moderator decides who will be involved and organises where and when a meeting will take place.

Question 5
Of the following, which is the most suitable test basis for component testing?

a. A detailed design.
b. A database module.
c. An interface definition.
d. A business process.

Question 6
Which of the following statements *best* describes the role of testing?

a. Testing ensures that the right version of code is delivered.
b. Testing can be used to assess quality.
c. Testing improves quality.
d. Testing shows that the software is error free.

Question 7
Which of the following *best* demonstrates the value of static testing?

a. Project A was late starting testing and subsequently overran the project target and budget.
b. Project B was late starting development because detailed requirements reviews were held but later completed on time and on budget.
c. Project C involved all testers in reviewing test specifications and subsequently completed late and over budget.
d. Project D overran because a defect occurred late in development, resulting in long delays.

Question 8
Which of the following characteristics will affect the test effort required to achieve the testing objectives of a project?

a. The development model in use.
b. The number of testers available.
c. The availability of suitable test environments.
d. The cost of required testing tools.

Question 9
Which of the following is a common test metric?

a. Number of testers in a test team.
b. Number of test cases prepared.
c. Number of test cases run/not run.
d. Number of requirements to be tested.

Question 10
Which of the following is not a main consideration for tool selection?

a. Evaluation of the tool against clear requirements and objective criteria.
b. Evaluating how the tool fits with existing processes and practices, and determining what needs to change.
c. Identification of opportunities for an improved test process supported by the tool.
d. Identification of internal requirements for coaching and mentoring in the use of the tool.

Question 11
Youngsters aged 12 are incorrectly issued with 'child' tickets on an adventure park website, instead of an 'adult' ticket. Which of the following describes this?

a. Failure.
b. Unprofessional.
c. Error.
d. Defect.

Question 12
Which of the following does not correctly describe how to derive test cases from a use case?

a. Use case tests can be designed to test all defined behaviour, including exceptional or alternative behaviour and error handling.
b. Use case testing is appropriate only for identifying behaviours that have not been defined.
c. Use case testing is used to ensure that all error handling is correctly executed.
d. Use case testing can be based on exceptional or alternative behaviours.

Question 13
Which of the following is generally true?

a. Code coverage is typically measured when carrying out functional testing.
b. Non-functional testing can make use of black-box techniques.
c. White-box testing focuses on system behaviours.
d. Functional testing is best done at system and acceptance testing.

Question 14
Which of the following statements about the purpose and content of a test plan is *not* correct?

a. The content of a master test plan must be completed before the project is initiated.
b. Test planning is a continuous activity and the content of test plans may change as the project progresses.
c. There may be separate test plans for different test types.
d. There is integration of test activities into the Software Development Life Cycle activities.

Question 15
Which of the following statements about decision coverage is correct?

a. Decision coverage is a measure of the number of white-box tests that have been executed.
b. Decision testing is the percentage of decision outcomes in a test object exercised by a suite of tests divided by the total number of decision outcomes in the test object.
c. Decision testing compares the number of decision outcomes executed by a suite of tests with the total number of tests.
d. Decision testing is the percentage of decision outcomes in a test object divided by the number of decision outcomes exercised by a suite of tests.

Question 16
Which of the following correctly explains error guessing?

a. Error guessing is based on testing for the types of mistakes that developers tend to make.
b. Error guessing is a technique for creating lists of mistakes, defects and failures that can be used by developers to avoid mistakes in the future.
c. Error guessing is based solely on the tester's experience of past mistakes.
d. Error guessing uses evidence from defect reports to ensure that the defect reported is no longer present.

Question 17
Which of the following is a valid entry criterion for a testing phase?

a. Testers are available to carry out the tests.
b. A suitable test environment is available.
c. All previous phases of testing have been completed.
d. The defects reported in previous phases have been cleared.

Question 18
Which of the following lists of activities are correctly sequenced for a product review process?

a. Distributing the work product, estimating effort and timescale, reviewing the work product, creating defect reports, analysing potential defects.
b. Estimating effort and timescale, distributing the work product, gathering metrics, analysing potential defects, creating defect reports.
c. Selecting reviewers, explaining scope and objectives, noting potential defects, evaluating quality characteristics, fixing defects found.
d. Identifying the review type, distributing the work product, analysing potential defects, fixing defects, communicating identified defects.

Question 19
What is a test oracle?

a. A person who knows most about testing within an organisation.
b. A means to identify expected results for specific test inputs.
c. A process to identify test conditions from source documents.
d. A means of estimating how long testing will take.

Question 20
Which of the following statements about exploratory testing is correct?

a. In exploratory testing, predefined tests are executed, logged and evaluated dynamically during test execution.
b. Exploratory testing is strongly associated with model-based test strategies and can incorporate other black-box techniques.

c. In exploratory testing, test results are used to learn more about the software under test and to create additional tests for areas that may need more testing.
d. Exploratory testing always uses session-based testing to ensure that tests are documented.

Question 21
Which of the following are part of test execution?

a. Identifying suitable test techniques, determining testing tasks and drawing up a test schedule.
b. Creating test suites from the test procedures, building the test environment and preparing test data.
c. Comparing actual results with expected results, running tests manually or using tools, and logging the outcome of tests run.
d. Examining the test basis, evaluating the quality of the test basis, and identifying and prioritising test conditions.

Question 22
What is the key difference between the metrics-based approach to estimation and the expert-based approach?

a. The metrics-based approach uses a mathematical equation to calculate total effort, whereas the expert-based approach uses data from previous projects.
b. The metrics-based approach uses data from previous projects, while the expert-based approach uses the experience of the tester doing the tests.
c. The expert-based approach uses the experience of testers, while the metrics-based approach uses previous estimates as a guide.
d. The expert-based approach uses the experience of experts, while the metrics-based approach uses data from former similar projects.

Question 23
Which of the following statements about checklist-based testing is correct?

a. Checklists are used to ensure that there is no variability in the actual testing.
b. Checklists can be used to support functional but not non-functional testing.
c. During analysis, testers may select an existing checklist and use it without modification or create a new checklist; modifications to existing checklists are not allowed.
d. In checklist-based testing, testers design, implement and execute tests to cover test conditions found in a checklist.

Question 24
Which of the following test tool types is classified as a test tool for test execution and logging and is also considered more appropriate for use by developers than by testers?

a. Model-based testing tools.
b. Test-driven development tools.
c. Test data preparation tools.
d. Coverage tools.

Question 25
Which of the following is a complete definition of how configuration management supports testing?

a. It ensures that testware and system components are uniquely identified.
b. It identifies and maintains the system components, the testware and the relationship between them.
c. It ensures that all identified documents are referenced unambiguously in test documentation.
d. It ensures that all test items are version controlled, tracked for changes and related to each other.

Question 26
A company rewards its sales people on the basis of their sales in each month. Sales staff who make sales worth more than £10,000 are paid the highest bonus, followed by those who earn more than £8,000, those who earn more than £5,000 and those who earn more than £3,000, with each group earning a bonus of 1 per cent less than the group above. Sales staff who earn £3,000 or less are paid no bonus for that month. What is the minimum number of test cases required to cover all valid equivalence partitions for calculating the sales bonus?

a. 6
b. 5
c. 4
d. 3

Question 27
Which of the following is the best reason to maintain traceability between the test basis and test work products?

a. Traceability ensures that all requirements have been tested.
b. Traceability assists in the impact analysis of potential changes.
c. Traceability enables the project to be delivered on time.
d. Traceability highlights who is to blame when defects are found.

Question 28
A central heating system timer is calibrated in hours and minutes, using a 24-hour clock. The system allows up to four time zones for each day. On a particular day the system is set to switch on from 06.00 to 08.45, from 11.45 to 13.15, and from 16.45 to 22.45. Using a two-point boundary value system, which of the following times are needed to test this day's functionality?

a. 06.00, 06.01, 08.45, 08.46, 11.45, 11.46, 13.15, 13.16, 16.45, 16.46, 22.45, 22.46
b. 05.59, 06.00, 08.44, 08.45, 11.44, 11.45, 13.14, 13.15, 16.44, 16.45, 2.244, 22.45
c. 06.00, 06.01, 08.44, 08.45, 11.44,11.45, 13.14, 13.15, 16.45, 16.46, 22.44, 22.45
d. 05.59, 06.00, 08.45, 08.46, 11.44, 11.45, 13.15, 13.16, 16.44, 16.45, 22.45, 22.46

Question 29
A requirements document for a new supermarket checkout system has been produced. It is 743 pages long and contains an overall description of the system, detailed work-flows and use cases, with mocked up screenshots of specific functions, and an outline of the proposed approach to development. Which of the following is a key success factor for a successful review of this document?

a. The review team must include designers, developers and testers but the document is too technical for users to review.
b. The document should be reviewed in small chunks and defects fed back to the authors as early as possible.
c. The review must take the form of an inspection so that detailed metrics are available.
d. The review should be arranged at the earliest possible opportunity so that the development team is not held up.

Question 30
Which of the following is a trigger for maintenance testing?

a. A new feature is required for an iteration.
b. A fix is required before the system goes live.
c. The system functionality has been descoped to hit a deadline.
d. Data is being migrated for a live system to a different platform.

Question 31
A system is being developed to manage the operation of a warehouse-based robot that collects items from shelves and takes them to a packaging area. The location in which the robot will operate also has human operators working, though in a separate area of the warehouse. How should a product risk analysis inform the testing?

a. By helping to determine the particular types and levels of testing to be performed.
b. By defining the specific safety features to be incorporated.
c. By mandating the use of particular test tools.
d. By mandating that all testers are experienced in safety-related systems.

Question 32
A university examination system sets all examinations for a module with the same criteria. A pass is awarded to any student who scores at least 40 per cent, a merit is awarded to any student scoring at least 60 per cent, and a distinction is awarded to any student scoring 85 per cent or over. Any student scoring at least 35 per cent but below 40 per cent is offered a viva voce exam and any student who scores at least 30 per cent but below 35 per cent is offered a resit. Any student scoring below 30 per cent is deemed to have failed the relevant module. All examination paper results are recorded as whole numbers. Which of the following sets of values needs to be identified for two-point boundary value analysis?

a. 0, 1, 30, 34, 35, 40, 41, 60, 61, 85, 86
b. 29, 30, 34, 35, 39, 40, 59, 60, 84, 85
c. 30, 31, 35, 36, 40, 41, 60, 61, 85, 86
d. 29, 30, 31, 34, 35, 36, 39, 40, 41, 59, 60, 61, 84, 85, 86

Question 33
A suite of tests has been run and some changes have been made to the relevant modules that have affected the priority of the tests.

The original priorities were as follows:

Test case	Priority
1	H
2	M
3	H
4	L
5	L
6	H

After the initial tests some remedial work was done, and this work has changed the dependencies between the test cases as follows:

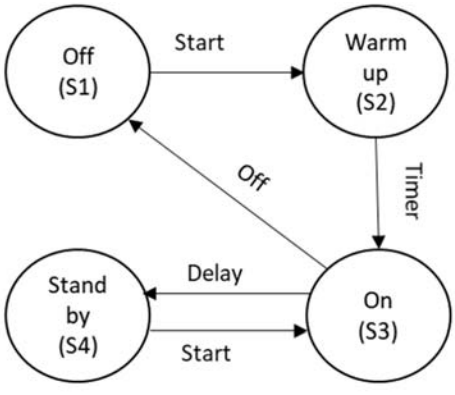
Test case 3 is now dependent on test case 5

Test case 2 is now dependent on test case 3

What should be the sequence of tests in the test execution schedule?

a. 1, 3, 6, 2, 4, 5
b. 1, 6, 5, 3, 2, 4
c. 1, 5, 3, 2, 6, 4
d. 1, 6, 3, 2, 5, 4

Question 34
Which of the statements about the state transition diagram and table of test cases is true?



Test case	1	2	3	4	5	6
Start state	S1	S2	S3	S4	S3	S4
Input	Start	Timer	Delay	Start	Off	Off
Expected end state	S2	S3	S4	S3	S1	S1

a. The given test cases cover all valid transitions and at least one invalid transition in the state transition diagram.
b. The given test cases test all valid transitions but not invalid transitions in the state transition diagram.
c. The given test cases test only some of the valid transitions in the state transition diagram.
d. The given test cases represent only some of the valid transitions and all invalid transitions in the state transition diagram.

Question 35
A defect report has been raised for a system that separates apples from a conveyor belt into different sizes and diverts them to appropriate belts. The test found that the system correctly channelled apples below a certain size but failed to detect larger apples and divert them to their own belt.

The dated defect report has a title and provides a short summary of the defect, the degree of risk involved and the severity allocated to the defect. It identifies the test script that was run and the expected result of the test, which was that apples above 10 cm in diameter should be diverted to belt 4.

Which of the following information, if any, is the *most* important omission from this defect report?

a. Recommendations.
b. Actual result.
c. Priority to fix.
d. Author.

Question 36
Requirements for a new supermarket checkout system are being reviewed by a team made up from checkout staff, developers, testers, technical support and management.

The requirements document provides detailed explanations of scenarios involving common problem situations, such as missing bar codes, unreadable labels, damaged goods and customers who find they have insufficient funds to pay the final bill.

It is important that the system meets all the needs of all the key stakeholders.

Which of the following review techniques is the *best* one to select to ensure that the right outcomes are achieved?

a. Checklist-based.
b. Scenarios and dry runs.
c. Ad hoc.
d. Perspective-based.

Question 37
Which of the following *best* describes the purpose of impact analysis for maintenance testing?

a. To determine if the test cases are out of date.
b. To determine if the current system should be changed.
c. To identify areas of code where maintainability might be an issue.
d. To identify areas of the testing process that could be improved.

Question 38
Which of the following do you want to find in the mindset of a tester?

i. Professional pessimism.
ii. Attention to detail.
iii. Assuming nothing will go wrong.
iv. The confirmation bias.
v. Ability to design solutions.

a. i and iii.
b. iv and v.
c. iii and iv.
d. i and ii.

Question 39
A vehicle insurance policy is subject to certain surcharges on the standard premium, added as percentages of the standard premium under certain conditions. The conditions are cumulative, so each condition that is met affects the premium.

Penalty points are applied for specific infringements of driving regulations and each offence can attract up to 3 penalty points. If a driver infringes more than one aspect of the driving regulations, only the highest penalty is applied.

The following decision table has been designed to test the logic for determining insurance premiums.

	T1	T2	T3	T4	T5	T6	T7	T8
Conditions								
3 penalty points	No	No	No	No	Yes	Yes	Yes	Yes
6+ penalty points	No	No	Yes	Yes	No	No	Yes	Yes
Age under 25	No	Yes	No	Yes	No	Yes	No	Yes
Action								
	Normal premium	+15%	+10%	+25%	+5%	+20%	+15%	+30%

Which test case must be eliminated because it is infeasible?

a. T1
b. T4
c. T5
d. T6

Question 40
Which of the following statements correctly describes the defect clustering testing principle?

a. Testing should be targeted at the most junior developer's code. This is where most defects will occur.
b. If no defects are found in the first 25 per cent of the time available, the code can be deemed safe to deliver to production.
c. Testing effort is to be targeted at what is thought to be the riskiest areas, and later those areas where there are more defects found.
d. Finding and fixing defects does not help if the system built is unstable and does not match user needs/expectations.

A2 MOCK CTFL EXAMINATION ANSWERS

Q1	c	Q11	a	Q21	c	Q31	a
Q2	b	Q12	b	Q22	d	Q32	b
Q3	c	Q13	b	Q23	d	Q33	b
Q4	a	Q14	a	Q24	d	Q34	a
Q5	a	Q15	b	Q25	b	Q35	b
Q6	b	Q16	a	Q26	b	Q36	d
Q7	b	Q17	b	Q27	b	Q37	b
Q8	a	Q18	c	Q28	d	Q38	d
Q9	c	Q19	b	Q29	b	Q39	b
Q10	b	Q20	c	Q30	d	Q40	c

A3 MOCK CTFL EXAMINATION COMMENTARY

Question 1
This is a K1 question relating to Learning Objective FL-1.5.1 – Identify the psychological factors that influence the success of testing.

Options a, b and d are all helping to build a 'them and us' culture between testers and developers, whereas option c is clearly mentioned in the syllabus as helping to lead to successful testing. This is therefore the required answer.

Question 2
This is a K1 question relating to the keywords from Chapter 4.

Option a is about creating tables to facilitate testing but is not related to testing of decision tables.

Option c is a different way of expressing the ideas in option a and is not about decision table testing.

Option d is about a white-box technique for testing conditions in code or other logic.

Decision table testing is a black-box technique related to the testing of decision tables as expressed in option b, which is taken from the glossary.

Question 3
This is a K1 question relating to Learning Objective FL-2.3.2 – Recognize that functional, non-functional, and white-box tests occur at any test level.

The correct answer is c, as confirmed by the glossary.

Option a is incorrect because some types of non-functional testing require specialist skills, such as performance testing, but not all.

Option b is also incorrect, because whilst some non-functional testing can be carried out by developers, such as looking for memory leaks, non-functional testing should also be carried out at the higher test levels.

Option d is incorrect because the coverage can be measured against the targets set.

Question 4
This is a K1 question relating to Learning Objective FL-3.2.2 – Recognize the different roles and responsibilities in a formal review.

Option b incorrectly allocates the task of assigning staff to the review leader (this is a management function).

Option c incorrectly suggests that reviewers must be subject matter experts, but reviewers can be drawn from any stakeholders as well as other specialists.

Option d incorrectly suggests that moderators decide who will be involved and time/place of a review (this are the review leader's responsibilities).

Option a correctly identifies the responsibilities of management as defined in the syllabus (section 3.2.2).

Question 5
This is a K2 question relating to Learning Objective FL-2.2.1 – Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities.

Option b – a database module – is incorrect; a database module could be a test object for component testing, but not a test basis.

Option c – an interface definition – is incorrect; this could be a test basis for integration testing.

Option d – a business process – is incorrect; this could be a test basis for acceptance testing.

Option a – a detailed design – is a suitable test basis for component testing.

Question 6
This is a K2 question relating to Learning Objective FL-1.2.1 – Give examples of why testing is necessary.

Testing cannot ensure that the right version of software being delivered – that is the task of configuration management – so option a is incorrect.

Testing, in itself, does not improve quality (option c). Testing can identify defects, but it is in the fixing of defects that quality is actually improved.

Testing cannot show that software is error free – it can only show the presence of defects (this is one of the seven testing principles), which rules out option d.

Testing can be used to assess quality; for example, by measuring defect density or reliability, so option b is correct.

Question 7
This is a K2 question relating to Learning Objective FL-3.1.2 – Use examples to describe the value of static testing.

Option a is a straightforward example of project delay; there is no indication of whether or not static testing was employed, so the value of static testing cannot be determined.

Option c indicates that test specifications were rigorously reviewed, but this did not prevent project overrun, so does not indicate the value of static testing.

Option d does not, in itself, demonstrate the value of static testing because it identifies a situation in which delay and overspend occurred, whether or not static testing was deployed.

Option b is the best option because it identifies a situation in which, although delay occurred early in the life cycle because static testing was deployed, the project still completed on time and on budget. This suggests that the use of static techniques may have made the development phase more efficient and effective.

Question 8
This is a K1 question relating to Learning Objective FL-5.2.5 – Identify factors that influence the effort related to testing.

Option b is incorrect because the overall effort required is not related to the number of testers. It may take longer with fewer testers, but the overall effort will be the same.

Option c is incorrect because, although testing cannot proceed without test environments, this does not affect the effort required to complete the testing.

Option d is incorrect because the effort required is not affected by the cost of any tools, though the testing budget may be.

Option a is correct because it will determine how much testing is needed at each stage and how many stages there will be.

Question 9
This is a K1 question relating to Learning Objective FL-5.3.1 – Recall metrics used for testing.

Option a is not a measure of testing, just a head count.

Option b is a count of how many test cases were written; it does not count test cases actually used in the tests.

Option d is a measure of the size of the test basis but not a metric of testing.

Option c is correct; it measures the total number of tests prepared and identifies how many were actually run and how many were not run.

Question 10
This is a K1 question relating to Learning Objective FL-6.2.1 – Identify the main principles for selecting a tool.

Options a, c and d are correct; all are taken directly from the syllabus, section 6.2.1.

Option b is drawn from syllabus section 6.2.2 and is about the use of pilot projects for introducing a tool into an organisation. Tool evaluation can only be achieved by using the tool in the organisation, so could not be part of the initial selection.

Question 11
This is a K2 question relating to Learning Objective FL-1.2.3 – Distinguish between error, defect, and failure.

Option b is incorrect because, while problems happen in software and software projects, even though work is undertaken by professionals, this does not in itself imply unprofessional behaviour. Option b really describes an attitude rather than what was done, and we have no way of knowing anything about the attitude of those working on this website.

An error is the underlying cause (perhaps the person writing the program specification misunderstood what was written in the requirements document) of a failure but not the failure itself, so option c is incorrect.

A defect may be the cause of a failure (e.g. perhaps the developer used '>' rather than '>=' in a condition in the code) but is not the actual failure, so option d is incorrect.

Incorrect tickets being issued is an observed consequence, which is the failure itself – so option a is the correct answer.

Question 12
This is a K2 question relating to Learning Objective FL-4.2.5 – Explain how to derive test cases from a use case.

Options a, c and d are all derived from the syllabus, section 4.2.5, and accurately describe how some aspect of use case testing is carried out.

Behaviours that have not been defined could not be systematically tested by use case testing, so option b is the correct answer because it does **not** describe how to derive test cases from a use case.

Question 13
This is a K2 question relating to Learning Objective FL-2.3.1 – Compare functional, non-functional, and white-box testing

Option a is incorrect because code coverage is normally measured in white-box testing, not when carrying out functional testing.

Option c is incorrect because white-box testing focuses on program behaviour, not system behaviour.

Option d is incorrect because functional testing should be done at all levels, not just at system and acceptance testing.

Option b is correct because non-functional testing can, and usually does, make use of black-box techniques.

Question 14
This is a K2 question relating to Learning Objective FL-5.2.1 – Summarize the purpose and content of a test plan.

Options b, c and d are all directly lifted from the syllabus, section 5.2.1.

Option a incorrectly states that a master test plan must be completed before a project starts. This explicitly contradicts the correct statement in option b. A master test plan may be used, with separate test plans for different test levels or test types, but since these can be updated throughout the project the master test plan must also change.

Question 15
This is a K2 question relating to Learning Objective FL-4.3.2 – Explain decision coverage.

Option a is incorrect because it measures only the number of white-box tests executed and not the coverage achieved.

Option c is incorrect because it calculates the number of decision outcomes achieved per test rather than the coverage achieved.

Option d is incorrect because it measures the inverse of decision coverage.

Option b correctly defines decision coverage in line with section 4.3.2 of the syllabus.

Question 16
This is a K2 question relating to Learning Objective FL-4.4.1 – Explain error guessing.

Option b is incorrect because, while such lists may be of value in avoiding defects, they are not used in error guessing.

Option c is incorrect because, while error guessing may be based partly on an individual tester's experience the technique utilises other sources, such as failures that have occurred in other applications.

Option d is incorrect because this is an example of retesting, not error guessing.

Option a is correct and reflects the syllabus, section 4.4.1.

Question 17
This is a K2 question relating to Learning Objective FL-5.2.3 – Give examples of potential entry and exit criteria.

Option a is a project management issue and not an entry criterion for a testing phase.

Option c is not necessarily relevant, since some earlier testing phases may not have delivered components or sub-systems for the part of the system about to be tested.

Option d is incorrect because not all defects reported in previous phases may be relevant, and clearance of defects that are relevant should be addressed by exit criteria from previous phases.

Option b is correct because testing cannot begin until a test environment is available.

Question 18
This is a K2 question relating to Learning Objective FL-3.2.1 – Summarize the activities of the work product review process.

Option a incorporates two planning activities, omits the initiation phase, includes the reviewing phase and swaps the issue communication and analysis phase with the fixing and reporting phase.

Option b also includes two items from the planning phase and one correct item from the initiation phase, but then omits the individual review phase before moving on to issue communication and analysis and fixing defects phases.

Option d correctly includes items from the planning phase, the initiation phase and the individual review phase, but then moves straight to fixing defects before the details of defects have been communicated (part of 'issue communication and analysis').

Option c is correct because it includes one item from each phase in the correct sequence.

Question 19
This is a K1 question relating to Learning Objective Keyword

Option a identifies a test expert or test guru, which is not the same thing as a test oracle, so option a is incorrect.

Option c encapsulates the process of test analysis, so option c is incorrect.

Option d is a test estimation method, so option d is incorrect.

Option b is clearly defined in the syllabus and is the correct answer.

Question 20
This is a K2 question relating to Learning Objective FL-4.4.2 – Explain exploratory testing.

Option a is incorrect because exploratory testing does not use predefined tests.

Option b is incorrect because exploratory testing is not associated with model-based test strategies but may sometimes be associated with reactive test strategies.

Option d is incorrect because, while exploratory testing may sometimes use session-based testing, this is not characteristic of exploratory testing. When session-based testing is used, this is to structure the testing activity rather than to ensure that tests are documented.

Option c is correct and corresponds to a statement in section 4.4.2 of the syllabus.

Question 21
This is a K2 question relating to Learning Objective FL-1.4.2 – Describe the test activities and respective tasks within the test process.

All of the options list activities from **one** of the groups of testing activities, so the task is to identify activities for test execution.

Option a is incorrect because it lists test-planning activities.

Option b is incorrect because it lists test implementation activities.

Option d is incorrect because it lists test analysis activities.

Option c is correct because it lists test execution activities.

Beware of option b – the activities listed are **preparation** for test execution, rather than for the running of tests themselves.

Question 22
This is a K2 question relating to Learning Objective FL-5.2.6 – Explain the difference between two estimation techniques: the metrics-based technique and the expert-based technique

Option a is incorrect because the use of a mathematical equation is not necessarily based on data from previous projects as required by the metrics approach, and the expert-based approach does not necessarily use data from previous projects.

Option b is incorrect because the expert-based approach requires an expert or at least an experienced tester.

Option c is incorrect because it bases metrics on previous estimates rather than on data about what actually happened.

Option d is correct as defined in the syllabus, section 5.2.6.

Question 23
This is a K2 question relating to Learning Objective FL-4.4.3 – Explain checklist-based testing.

Option a is incorrect because checklists are high-level lists and some variability in testing is likely to occur.

Option b is incorrect because checklists can be used to support a wide variety of test types.

Option c is incorrect because testers may create new checklists, expand exiting checklists or use an existing checklist.

Option d is correct as defined in section 4.4.3 of the syllabus.

Question 24
This is a K2 question relating to Learning Objective FL-6.1.1 – Classify test tools according to their purpose and the test activities they support.

Option a is incorrect because model-based testing tools are categorised as tools for test design and implementation and they are suitable for use by testers.

Option b is incorrect because test-driven development tools are best suited to developers but are categorised as tool support for test design and implementation.

Option c is incorrect because test data preparation tools are categorised as tool support for test design and implementation and are suitable for use by testers.

Option d is the correct answer because coverage tools are classified as tool support for test execution and logging and are most suitable for developers.

Question 25
This is a K2 question relating to Learning Objective FL-5.4.1 – Summarise how configuration management supports testing.

Option a is correct but incomplete because it mentions only the testware and system components separately and there is no mention of the relationships between items.

Option c is correct but incomplete because it relates only to documentation.

Option d is incorrect because it relates only to test items.

Option b correctly refers to managing the system components, the testware and the relationship between them.

Question 26
This is a K3 question relating to Learning Objective FL-4.2.1 – Apply equivalence partitioning to derive test cases from given requirements

The required partitions are:

- > £10,000
- £8,001–£10,000
- £5,001–£8,000
- £3,001–£5,000
- < £3,001

There are five partitions, so option b is correct.

Question 27
This is a K2 question relating to Learning Objective FL-1.4.4 – Explain the value of maintaining traceability between the test basis and test work products.

Option a is incorrect. This could help in clarifying whether all requirements are covered by one or more test cases, but the requirements could be included in a test case that was not actually run.

Option c is incorrect because traceability has no direct bearing on whether the project will be delivered on time, and it is not something that is stated (in the syllabus or elsewhere) as a 'benefit' of traceability.

Option d is incorrect and inappropriate in that it implies a 'blame culture' rather than one where cooperation and product quality are to the forefront.

Option b is correct and is specifically mentioned in section 1.4.4 of the syllabus.

Question 28
This is a K3 question relating to Learning Objective FL-4.2.2 – Apply boundary value analysis to derive test cases from given requirements.

Option a is incorrect because it starts each time zone on the boundary rather than just under, but ends each time zone correctly.

Option b is incorrect because it starts each time zone correctly but does not check the end of each time zone correctly.

Option c is incorrect because it starts each time zone on the boundary rather than before the boundary and ends each time zone too early.

Option d is correct because it correctly tests the values just before and on the lower boundaries, and on and just over the higher boundaries.

Question 29
This is a K2 question relating to Learning Objective FL-3.2.5 – Explain the factors that contribute to a successful review.

Option a is incorrect: it is always important for users to review requirements documents; the presence of designers, developers and testers may help with any technical terms.

Option c is incorrect: an inspection is not likely to be appropriate for this document and one key success factor is that an appropriate review type is applied. Metrics are not important at this stage, but removal of ambiguity, clarity of expression and understandability for users are vital.

Option d is incorrect: reviews should always be scheduled with adequate notice and time for participants to prepare.

Option b is the correct answer (in line with section 3.2.5 in the syllabus): it will allow reviews to begin earlier and will provide earlier feedback to authors to enable improvements to be made continually.

Question 30
This is a K2 question relating to Learning Objective FL-2.4.1 – Summarize triggers for maintenance testing.

Option a is incorrect because a new feature required for an iteration implies development rather than maintenance.

Option b is incorrect because it again implies development rather than maintenance.

Option c is incorrect for the same reason.

Option d is correct because data migration is a typically maintenance activity.

Question 31
This is a K2 question relating to Learning Objective FL-5.5.3 – Describe, by using examples, how product risk analysis may influence the thoroughness and scope of testing.

Options b, c and d are incorrect because they are all related to managing the project.

Option a specifically relates to the testing activities and how these need to be driven by risk levels.

Question 32
This is a K3 question relating to Learning Objective FL-4.2.2 – Apply boundary value analysis to derive test cases from given requirements.

Option a is incorrect because it incorrectly tests for 0 and 1 as well as incorrectly identifying boundary values for partitions.

Option c is incorrect because it tests the lower boundaries incorrectly (though it tests the upper boundaries correctly).

Option d incorrectly uses three values at each boundary.

Option b is correct, testing below and on the lower boundaries and on and above the upper boundaries.

Question 33
This is a K3 question relating to Learning Objective FL-5.2.4 – Apply knowledge of prioritization, and technical and logical dependencies, to schedule test execution for a given set of test cases.

Option a was the original schedule, but this has now been altered, so option a is incorrect.

Option c places test cases 5, 3 and 2 in the correct sequence but test case 6, which is high priority, is relegated to fifth place, so option c is incorrect.

Option d places 6 in second position but does not make test case 3 dependent on test case 5, so option d is incorrect.

Option b is the correct answer because it correctly sequences 5, 3 and 2 and places test case 6 ahead of this trio.

Question 34
This is a K3 question relating to Learning Objective FL-4.2.4 – Apply state transition testing to derive test cases from given requirements.

Option b is incorrect because an invalid transition is represented (test case 6).

Option c is incorrect because all of the valid transitions are represented (test cases 1–5 correspond to the five valid transitions shown in the diagram).

Option d is incorrect because all valid transitions are represented, and one invalid transition is addressed in test case 6.

Option a is correct because the table correctly identifies the five valid transitions (test cases 1–5) and one invalid transition (test case 6).

Question 35
This is a K3 question relating to Learning Objective FL-5.6.1 – Write a defect report, covering defects found during testing.

Options a, c and d are all incorrect because they list valid fields on a defect report, but none of them prevents action being taken to resolve the problem.

Option b is the correct answer because the actual result is needed to enable corrective action to be correctly applied.

Question 36
This is a K3 question relating to Learning Objective FL-3.2.4 – Apply a review technique to a work product to find defects.

Option a is not the best answer. Checklists can help to focus attention on specific aspects of a system and to ensure that typical defect types are addressed, but this is not an ideal mechanism for addressing defects that will affect multiple stakeholders.

Option b is a better option, in that it provides specific scenarios for reviewers to consider, but it still does not provide the breadth of involvement required.

Option c is unlikely to be effective in that it provides little or no guidance to reviewers.

Option d is the best answer because it encourages individual reviewers to take on multiple stakeholder viewpoints. This makes the overall review more effective and could be modified to enable reviewers to cooperate in working through scenarios incorporated into the requirements document (and possibly also identify new scenarios to be considered).

Question 37
This is a K2 question relating to Learning Objective FL-2.4.2 – Describe the role of impact analysis in maintenance testing.

Option a is incorrect because out-of-date test cases are a hindrance to maintenance, but this is not the purpose of impact analysis.

Option c is incorrect because achieving maintainability is an issue for development, not maintenance.

Option d is incorrect; this might be part of a retrospective or post-implementation review but is not related to impact analysis.

Option b is correct because a decision should be based on the likely consequences of the change, which is one purpose of impact analysis. Impact analysis is also about determining what testing will be needed following a change.

Question 38
This is a K2 question relating to Learning Objective FL-1.5.2 – Explain the difference between the mindset required for test activities and the mindset required for development activities.

All of the choices are mentioned in section 1.5.2 of the syllabus – but not all in a positive light from the perspective of testers.

Two of the choices (i and ii) are listed as favourable for testers. Items iii and iv in the list are possible aspects of a developer's mindset that are detrimental to a tester perspective. Item v is a positive component of a developer mindset that has no particular bearing on testing.

Option d is the correct answer because it is the only option that mentions items i and ii.

Question 39
This is a K3 question relating to Learning Objective FL-4.2.3 – Apply decision table testing to derive test cases from given requirements.

Option a is feasible and will apply to most applicants, so it is not the correct answer.

Options c and d both involve penalising for 3 points and for 6+ points, which is valid within the rules, so neither of these is the correct answer.

Option b is infeasible because it is not possible to incur 6 penalty points without also incurring 3 penalty points.

Option b is therefore the correct answer.

Be careful in questions like this one to note that the question asked for **infeasible** test cases. It is easy, especially under time pressure, to opt for the more usual expectation of identifying feasible test cases.

Question 40
This is a K2 question relating to Learning Objective FL-1.3.1 – Explain the seven testing principles.

Option a appears to reflect the defect clustering principle, but there is no reason to assume that defect clustering will be associated with the work of the most junior developer, so option a is incorrect.

Option b is an example of the 'absence of errors' fallacy – the fact that no errors have been found does not mean that code can be released into production – so option b is incorrect.

Option d is a direct statement of the 'absence of errors' fallacy, so option d is incorrect.

Option c is the best statement of the defect clustering principle – defects are often found in the same places, so testing should focus first on areas where defects are expected or where defect density is high. Option c is therefore the correct answer.

INDEX

SOFTWARE TESTING

An ISTQB-BCS Certified Tester Foundation guide
Fourth edition

Brian Hambling (editor)

Updated in line with the 2018 ISTQB Certified Tester Foundation Level (CTFL) Syllabus, this best-selling software testing title explains the basic steps of software testing and how to perform effective tests. It provides an overview of different techniques, both dynamic and static. It is the only official textbook of the ISTQB-BCS Certified Tester Foundation Level, with self-assessment exercises, guidance notes on the syllabus topics and sample examination questions.

This fourth edition includes examples and exercises reflecting current technology and applications. It is ideal for those with a little experience of software testing who wish to cement their knowledge with industry-recognised techniques and theory.

- **BCS' bestselling software testing title**
- **The only official textbook of the ISTQB-BCS Certified Tester Foundation Level certificate**
- **Updated with 2018 syllabus content, standards and example test questions**

ABOUT THE AUTHORS

The authors are all experienced BCS examination setters and markers and know the ISTQB syllabus in depth. The editor, Brian Hambling, has experienced software development from a developer's, project manager's and quality manager's perspective in a career spanning over 35 years.

You might also be interested in:

This book covers all the sections of the latest 2018 CTFL syllabus and more. It is not just written as an exam aid though, it is a reference for software testing in its own right...for anyone involved in the development of software, whatever development methodology the project follows.

Phil Isles, Test Manager, private banking

A jewel of a book; direct, clear, and to the point. The text is no longer than the necessary minimum, it's pedagogical, and has good exercises. Every sentence has a purpose.

Ole Einar Arntzenon, Taranaki AS
Review of previous edition

Invaluable for anyone involved in testing and would lift the game of most VV&T staff.

IT Training Magazine
Review of previous edition

Information Technology

Cover photo: iStock © mevans

ISBN 978-1-78017-492-1

bcs The Chartered Institute for IT

Paperback available