

# ANNE METTE JONASSEN HASS

```
* Execute commands
* @access public
* @return boolean $bResult
}

function deleteRegisterAssistance($nidBenefit, $percentage, $nsystem)
$bResult = $oRegisterAssistanceBD->deleteAssistance($nidBenefit);
if($bResult) {
    $oStructureTransaction = new StructureTransaction();
}

function startDataAssistance($nidBenefit, $percentage, $nsystem)
UserPercentager, $sobs);
$oDataAssistance = new DataAssistance($nidBenefit, $percentage, $nsystemUserP
ercentager, $sobs);
return $oDataAssistance;
```

# GUIDE TO ADVANCED SOFTWARE TESTING

```
* Execute commands
* @access public
* @return boolean $bResult
}

function deleteRegisterAssistance($nidBenefit, $percentage, $nsystem)
$bResult = $oRegisterAssistanceBD->deleteAssistance($nidBenefit);
if($bResult) {
    $oStructureTransaction = new StructureTransaction();
}

function startDataAssistance($nidBenefit, $percentage, $nsystem)
UserPercentager, $sobs);
$oDataAssistance = new DataAssistance($nidBenefit, $percentage, $nsystemUserP
ercentager, $sobs);
```

# **Guide to Advanced Software Testing**

For a listing of recent related Artech House titles,  
please turn to the back of this book.

# **Guide to Advanced Software Testing**

Anne Mette Jonassen Hass



**ARTECH  
HOUSE**

BOSTON | LONDON  
[artechhouse.com](http://artechhouse.com)

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

A catalog record for this book is available from the U.S. Library of Congress.

**British Library Cataloguing in Publication Data**

A catalogue record for this book is available from the British Library.

ISBN-13: 978-1-59693-285-2

Cover design by Yekaterina Ratner

© 2008 ARTECH HOUSE, INC.

685 Canton Street  
Norwood, MA 02062

All rights reserved. Printed and bound in the United States of America. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Artech House cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

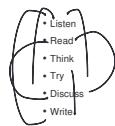
10 9 8 7 6 5 4 3 2 1

*To the most important women in my life:  
my grandmother, Martha,  
my mother, Alice,  
my sister, Lene,  
and my daughter, Lærke*

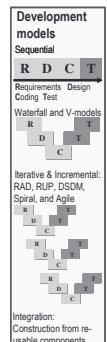


# Contents

Foreword.....	xv
Preface.....	xvii
<b>I</b>	
A Guide to Advanced Testing .....	xix
I.1 Reading Guidelines .....	xx
I.2 Certified Tester, Advanced Level .....	xx
I.2.1 This Book in Relation to the Syllabus .....	xxi
I.2.2 Ethics for Testers .....	xxvi
I.3 Software Testing Basics.....	xxvii
I.3.1 Terms and Definitions in Testing .....	xxvii
I.3.2 Testing Is Multidimensional .....	xxviii
I.3.3 Definition of Testing .....	xxix
Questions.....	xxx
Appendix IA Vignettes .....	xxxi

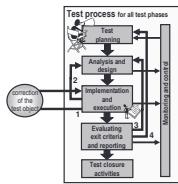


<b>1</b>	Basic Aspects of Software Testing .....	1
1.1 Testing in the Software Life Cycle .....	1	
1.1.1 Development Models .....	2	
1.1.2 Dynamic Test Levels .....	8	
1.1.3 Supporting Processes .....	16	
1.2 Product Paradigms .....	23	
1.2.1 Systems of Systems .....	24	
1.2.2 Safety-Critical Systems .....	25	
1.3 Metrics and Measurement.....	28	
1.3.1 Measuring in General .....	28	
1.3.2 Test-Related Metrics .....	29	
1.3.3 Analysis and Presentation of Measurements .....	31	
1.3.4 Planning Measuring .....	31	
Questions.....	31	

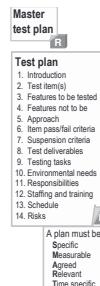


<b>2</b>	Testing Processes .....	33
2.1 Processes in General.....	34	
2.1.1 The Concept of a Process .....	34	

2.1.2 Monitoring Processes	34
2.1.3 Processes Depend on Each Other	35
2.1.4 The Overall Generic Test Process	35
2.1.5 Other Testing Processes	39
<b>2.2 Test Planning and Control .....</b>	<b>39</b>
2.2.1 Input to Test Planning and Control	40
2.2.2 Documentation of Test Planning and Control	41
2.2.3 Activities in Test Planning	41
2.2.4 Activities in Test Control	50
2.2.5 Metrics for Test Planning and Control	50
<b>2.3 Test Analysis and Design .....</b>	<b>50</b>
2.3.1 Input to Test Analysis and Design	51
2.3.2 Documentation of Test Analysis and Design	51
2.3.3 Activities in Test Analysis and Design	51
2.3.4 Requirements	57
2.3.5 Traceability	60
2.3.6 Metrics for Analysis and Design	61
<b>2.4 Test Implementation and Execution .....</b>	<b>61</b>
2.4.1 Input to Test Implementation and Execution	62
2.4.2 Documentation of Test Implementation and Execution	62
2.4.3 Activities in Test Implementation and Execution	62
2.4.4 Metrics for Implementation and Execution	71
<b>2.5 Evaluating Exit Criteria and Reporting.....</b>	<b>71</b>
2.5.1 Input to Test Progress and Completion Reporting	72
2.5.2 Documentation of Test Progress and Completion Reporting	72
2.5.3 Activities in Test Progress and Completion Reporting	72
2.5.4 Metrics for Progress and Completion Reporting	73
<b>2.6 Test Closure .....</b>	<b>74</b>
2.6.1 Input to Test Closure	74
2.6.2 Documentation of Test Closure	74
2.6.3 Activities in Test Closure	75
2.6.4 Metrics for Test Closure Activities	76
<b>Questions.....</b>	<b>76</b>



<b>3</b>	Test Management.....	79
3.1	Business Value of Testing.....	79
3.1.1	Purpose of Testing.....	80
3.1.2	The Testing Business Case.....	81
3.2	Test Management Documentation.....	85
3.2.1	Overview.....	85
3.2.2	Higher Management Documentation.....	86
3.2.3	Project Level Test Management Documentation.....	96
3.3	Test Estimation.....	106
3.3.1	General Estimation Principles.....	106
3.3.2	Test Estimation Principles.....	107
3.3.3	The Estimation Process.....	108
3.3.4	Estimation Techniques.....	109
3.3.5	From Estimations to Plan and Back Again.....	114
3.3.6	Get Your Own Measurements.....	115
3.4	Test Progress Monitoring and Control .....	115
3.4.1	Collecting Data.....	116
3.4.2	Presenting the Measurements.....	116
3.4.3	Stay in Control.....	124
3.5	Testing and Risk .....	125
3.5.1	Introduction to Risk-Based Testing.....	125
3.5.2	Risk Management.....	131
3.5.3	Risk Analysis.....	135
3.5.4	Risk Mitigation.....	142
	Questions.....	147



<b>4</b>	Test Techniques.....	151
4.1	Specification-Based Techniques .....	152
4.1.1	Equivalence Partitioning and Boundary Value Analysis.....	152
4.1.2	Domain Analysis.....	160
4.1.3	Decision Tables.....	166
4.1.4	Cause-Effect Graph.....	169
4.1.5	State Transition Testing.....	173

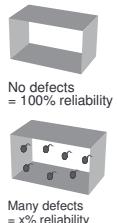


4.1.6 Classification Tree Method	179
4.1.7 Pairwise Testing	186
4.1.8 Use Case Testing	191
4.1.9 Syntax Testing	193
	
4.2 Structure-Based Techniques .....	197
4.2.1 White-Box Concepts	198
4.2.2 Statement Testing	199
4.2.3 Decision/Branch Testing	201
4.2.4 Condition Testing	202
4.2.5 Multiple Condition Testing	204
4.2.6 Condition Determination Testing	205
4.2.7 LCSAJ (Loop Testing)	206
4.2.8 Path Testing	209
4.2.9 Intercomponent Testing	210
4.3 Defect-Based Techniques .....	211
4.3.1 Taxonomies	211
4.3.2 Fault Injection and Mutation	213
	
4.4 Experience-Based Testing Techniques.....	214
4.4.1 Error Guessing	215
4.4.2 Checklist-Based	216
4.4.3 Exploratory Testing	218
4.4.4 Attacks	221
4.5 Static Analysis .....	222
4.5.1 Static Analysis of Code	223
4.5.2 Static Analysis of Architecture	230
4.6 Dynamic Analysis.....	233
4.6.1 Memory Handling and Memory Leaks	233
4.6.2 Pointer Handling	234
4.6.3 Coverage Analysis	234
4.6.4 Performance Analysis	235
4.7 Choosing Testing Techniques.....	235
4.7.1 Subsumes Ordering of Techniques	236
4.7.2 Advice on Choosing Testing Techniques	236
Questions.....	237

Appendix 4A Classification Tree Example .....	241
---	-----

<b>5</b> Testing of Software Characteristics .....	243
--	-----

5.1 Quality Attributes for Test Analysts .....	244
5.1.1 Functional Testing	245
5.1.2 Usability Testing	249
5.2 Quality Attributes for Technical Test Analysts.....	254
5.2.1 Technical Testing in General	256
5.2.2 Technical Security Testing	258
5.2.3 Reliability Testing	261
5.2.4 Efficiency Testing	265
5.2.5 Maintainability Testing	268
5.2.6 Portability Testing	271
Questions.....	273

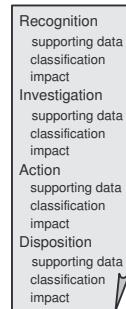


<b>6</b> Reviews (Static Testing).....	275
--	-----

6.1 General Principles for Static Testing .....	275
6.1.1 History of Static Testing	275
6.1.2 Static Testing Definition	276
6.1.3 Static Testing Cost/Benefit	278
6.1.4 Static Testing Generic Process	279
6.1.5 Roles in Static Testing	281
6.1.6 Static Testing Type(s) Selection	282
6.2 Static Testing Types .....	284
6.2.1 Informal Review	284
6.2.2 Walk-Through	285
6.2.3 Technical Review	286
6.2.4 Management Review	288
6.2.5 Inspection	289
6.2.6 Audit	300
6.3 Static Testing in the Life Cycle .....	301
6.4 Introducing Static Testing.....	303
6.4.1 Static Testing Implementation Roles	303
6.4.2 Static Testing Processes	304



6.4.3 Static Testing Piloting	305	
6.4.4 Static Testing Rollout	305	
6.4.5 Psychological Aspects of Static Testing	306	
Questions.....	306	
Appendix 6A Solution to the Flower Drawing.....	309	
<b>7</b>	<b>Incident Management.....</b>	<b>311</b>
7.1 Incident Detection.....	311	
7.1.1 Incident Definition	311	
7.1.2 Incident Causes	312	
7.1.3 Incident Reporting and Tracking	312	
7.2 Incident and Defect Life Cycles .....	313	
7.2.1 Incident Recognition	314	
7.2.2 Incident Investigation	315	
7.2.3 Incident Action	317	
7.2.4 Incident Disposition	318	
7.3 Incident Fields.....	319	
7.4 Metrics and Incident Management.....	319	
7.5 Communicating Incidents .....	321	
Questions.....	322	
Appendix 7A Standard Anomaly Classification.....	324	
Appendix 7B Change Control Process .....	327	
<b>8</b>	<b>Standards and Test Improvement Process .....</b>	<b>329</b>
8.1 Standards.....	330	
8.1.1 Standards in General	330	
8.1.2 International Standards	331	
8.1.3 National Standards	332	
8.1.4 Domain-Specific Standards	332	
8.2 Test Improvement Process .....	333	
8.2.1 Process Improvement Principles	334	
8.2.2 Process Maturity Models in General	337	
8.2.3 Testing Improvement Models	341	
Questions.....	357	



Appendix 8A Definition of Levels in the TPI Model .....	358
---	-----

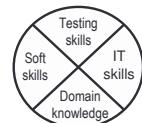
<b>9</b> Testing Tools and Automation .....	361
---	-----

9.1 Testing Tool Acquisition .....	362
9.1.1 Tool or No Tool? .....	362
9.1.2 Tool Selection Team .....	363
9.1.3 Testing Tool Strategy .....	363
9.1.4 Preparation of a Business Case .....	363
9.1.5 Identification of Tool Requirements .....	364
9.1.6 Buy, Open-Source, or Do-It-Yourself .....	365
9.1.7 Preparation of a Shortlist of Candidates .....	366
9.1.8 Detailed Evaluation .....	366
9.1.9 Performance of Competitive Trials .....	367
9.2 Testing Tool Introduction and Deployment .....	367
9.2.1 Testing Tool Piloting .....	368
9.2.2 Testing Tool Rollout .....	369
9.2.3 Testing Tool Deployment .....	369
9.3 Testing Tool Categories .....	370
9.3.1 Testing Tool Classification .....	370
9.3.2 Tools for All Testers .....	371
9.3.3 Tools for Test Analysts and Technical Test Analysts .....	373
9.3.4 Tools for Technical Test Analysts .....	380
9.3.5 Tools for Programmers .....	382
Questions .....	383
Appendix 9A List of Testing Tools .....	385

<b>10</b> People Skills .....	389
-------------------------------	-----

10.1 Individual Skills .....	389
10.1.1 Test Roles and Specific Skills .....	391
10.1.2 Testing by Other Professionals .....	392
10.1.3 Interpersonal Skills .....	392
10.2 Test Team Dynamics .....	394
10.2.1 Team Roles .....	395
10.2.2 Forming Testing Teams .....	397

A fool with a tool  
is still a fool





10.3 Fitting Testing in an Organization .....	398
10.3.1 Organizational Anchorage .....	398
10.3.2 Independence in Testing .....	399
10.4 Motivation .....	401
10.4.1 Maslow's Pyramid of Needs .....	402
10.4.2 Herzberg's Factors .....	403
10.4.3 K. B. Madsen's Motivation Theory .....	404
10.4.4 Testers' Motivation .....	405
10.5 Team Communication .....	405
Questions.....	407
Selected Bibliography.....	409
About the Author .....	411
Index .....	413

## **Foreword**

Years ago when I approached software development, my own sense of self-importance and infallibility was such that I did not feel like buying a book such as this. In those days, software had different constraints and was less complex than what is currently delivered now.

Nowadays, with all the interactions, interoperability, and dependencies expected between programs, the portability and internationalization expected by the users, among others, make this profession of software developer much more complex and challenging.

This challenge gave rise to new profiles such as those of the test manager, test analyst, and technical test analyst, who together with the development teams must ensure that developed software fits the expectations of the different stakeholders, whether project managers, business analysts, or end users.

To validate the level of knowledge associated with these software tester profiles, the ISTQB proposed a tester certification scheme built of two main levels: foundation and advanced with a syllabus for each. This book provides additional information to those available in these syllabi.

If I were to start developing or testing software today, I hope that my sense of self importance would give way before this book, and that I would give its content more than a glance or two, and seriously ponder the different aspects explained in its pages.

As the person responsible for the working party who published the ISTQB advanced-level syllabus in 2007, I am sure that this book will serve as a training support and reference for a number of future ISTQB advanced level certified testers.

The explanations, examples, and exercises provided in this book will allow you to understand the intricacies of testing and help you attain the expected proficiency to claim the ISTQB advanced-level tester certification.

*B. Homès  
Chair of Advanced Level Working Party  
Founder and principal consultant for TESSCO Technologies inc.  
Ollioules, France  
March 2008*



## Preface

“Write a test book? Never!” This was my position for many years, when the thought occurred to me or when colleagues or course delegates suggested it. I had and still have great respect for all the very good testing books already out there.

No book, however, seemed to cover the entire syllabus for the ISEB practitioner certification when I took that, nor when I started to train practitioners-to-be. I therefore wrote small bits of notes for my delegates to illustrate and expand on the topics in the syllabus. The notes just grew on me and in the end I had more than 500 pages.

When the new ISTQB advanced level syllabus began to appear I started to rewrite my notes to fit with that—and the book was born. I have followed the ISTQB advanced level syllabus closely because I find that the structure is strong and makes sense, and because the “notes” were intended for my own training courses based on the syllabus. Such a closely defined job was a challenge in some places I really had to take my own view on things and shake it about; sometimes it came out OK, and sometimes I got wiser.

The ISTQB advanced level syllabus is based on some of the best of the existing testing books. This book does not pretend to be better or truer. No book, no course, no person can provide the truth about testing. The book is intended as another voice in the constant dialogue going on between people with an interest in testing where thoughts and ideas are being exchanged. I hope the book will work as such, and as an inspiration and an aid to testers wanting to listen to yet another understanding of the testing subject, so difficult to get to grips with.

I also hope that it will help the promotion of the ISTQB certification, as I find this a great opportunity for testers to get a common language and work together to strengthen the understanding of testing in the entire software development industry.

A poster has been created to reflect the contents of this book. It is available at [www.deltaaxiom.com/poster](http://www.deltaaxiom.com/poster). I very much like to make pictures, both on paper and in my head. One of my aims when I teach is to help the delegates create a picture of what testing is about. Everybody’s picture is different, but after having drawn many pictures the images started to come together in the poster. It therefore shows a little bit of my brain, namely the bit where my present understanding of testing is.

I suggest you download the poster and use it and the book together: the poster to see an overview of the elements in testing and the book to go behind the picture and obtain more substance on the elements.

And remember:

- ▶ Testing is difficult.
- ▶ Testing requires overview.
- ▶ Testing requires creativity.
- ▶ Testing requires systematic work.
- ▶ Testing requires imagination.
- ▶ Testing requires courage.
- ▶ Testing is fun.

## Acknowledgments

Many people have contributed to this book. My boss Jørn Johansen gave me time and permission to write it. My colleagues, especially Carsten Jørgensen, gave me inspiration and plenty of their time to discuss all types of issues, great and small, and also contributed directly to the text. My former colleague Claus Lehmann-Lessél got me started on the poster. Two very professional test-ladies, Stine Laforce and Patricia Ensworth, reviewed the manuscript, and they did a fantastic job. My longtime friend Eddy Bøgh Brixen gave me graphic advice, not least regarding the poster, and he drew all the vignettes. Last but not least, I had the full support of my husband, Finn, and my daughter, Lærke, throughout the long days and weekends of writing. I am very grateful to you all.

There is no way I can mention all the people who have taught, inspired, and helped me during my testing career, be it managers, colleagues, developers, customers, tutors at courses, and speakers at conferences. Thanks to you all; I hope you know how much you mean to me!

## INTRODUCTION

# I

### A Guide to Advanced Testing

This book is a paradox. It is written for testers who want to become real advanced testing practitioners, but there is no way you can become a practitioner just by reading. A guide, however, can be good for preparing the journey and for help on the way.

This book is based on the ISTQB Certified Tester, Advanced Level Syllabus, Version 2007 Beta, and on the ISTQB Glossary of Terms used in Software Testing Version 1.2, April 2006, and the extension to the glossary included in the syllabus. ISTQB is the “International Software Testing Qualification Board,” an independent organization made up of member boards from more than 30 countries and regions around the world. See more at <http://www.istqb.org>.

The book can be used even if you don’t want to take an ISTQB advanced level certificate. In fact *the main purpose of the book is to inspire you to be an even better tester than you already are*; it is a guide for already experienced testers on their way to becoming truly professional testers. According to the *Collins Pocket English Dictionary*, a professional is engaged in and worthy of the standards of an occupation requiring advanced education!

A professional tester is a person who puts test knowledge into action in a professional way. He or she must have knowledge and understanding of the basics of testing, and some experience in deploying the knowledge in testing practice. An advanced education goes further than knowledge and understanding and aims at providing the tester with abilities to analyze complete and complex test assignments.

The ISTQB advanced certification is further explained later; that section can be skipped if you do not intend to be certified.

The basic philosophy of (software) testing is also discussed below. That section should be, if not read, then at least skimmed, to brush up on the foundation of testing.

### Contents

- I.1 Reading Guidelines
- I.2 Certified Tester, Advanced Level
- I.3 Software Testing Basics

*Do not forget that testing is not natural science.* There is no absolute solution to how it must be done; in fact there are many different schools and convictions for the approach to testing. This book represents one, mine in combination with that expressed by ISTQB. You will find that you agree and disagree as you read; the important thing is for you to find out what you believe to be the “right” way.

## I.1 Reading Guidelines

This book contains this basic introduction and 10 chapters, each covering a topic in the syllabus. The 10 chapters are:



1. Basic Aspects of Software Testing;
2. Testing Processes;
3. Test Management;
4. Test Techniques;
5. Testing of Software Characteristics;
6. Reviews (Static Testing);
7. Incident Management
8. Standards and Test Improvement Process;
9. Test Tools and Automation;
10. People Skills.

The chapters are structured in the same way:



- ▶ A very short appetizer to the contents of the chapter, including an overview of the sections in the chapter;
- ▶ The text;
- ▶ A list of questions, which may be used for repetition or as basis for discussions in a study group.

Some chapters have appendices with additional information. A number of vignettes are used in the margin to attract attention to specific information. These are explained in Appendix IA.

**Ex.**

Examples are marked in light gray.

## I.2 Certified Tester, Advanced Level

In the words of the ISTQB Certified tester, advanced level syllabus: “The advanced level certification is aimed at people who have achieved an advanced point in their careers in software testing. To receive advanced level certification, candidates must hold the foundation certificate and satisfy the exam board that they have sufficient practical experience to be considered advanced level qualified.”

To pass the exam, candidates must also demonstrate that they have achieved the learning objectives provided in the syllabus.

### I.2.1 This Book in Relation to the Syllabus

This book is based on the ISTQB Certified Tester Advanced Level Syllabus and the ISTQB Glossary of Terms used in Software Testing Version 1.2, April 2006.

*This book does in no way replace the syllabus in terms of what must be learned and understood for the certification. Where there are discrepancies between the syllabus and this book, the syllabus prevails! I cannot guarantee that only the ISTQB terms and definitions are used, or that any usage of a term is strictly in accordance with the glossary, though I have taken care.*

The structure of the book follows the structure of the syllabus to a very large extent. A few sections are placed differently and a few sections are left out, because the descriptions in the syllabus seem comprehensive.

This book, like the syllabus, is monolithic; that is, each topic is covered comprehensively in one place, even though the individual topics have different weights for the different paths in the certification scheme. The syllabus explains in detail what the learning objectives are for each topic for each of the certification paths. It is up to the reader to figure out which section to study extensively and which to skim or even skip.

The ISTQB software testing advanced level certification is a demanding professional education in testing, based on the ISEB/ISTEB software testing foundation certification.

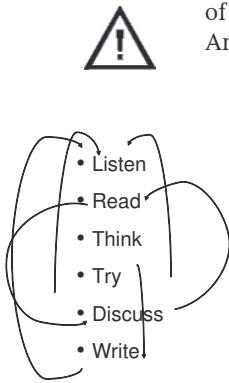
If we compare the testing education with getting a driver's license and driving a car, then the foundation certification is like getting the theoretical part and base the license on that and a little bit of supervised practice. You have to learn all the traffic rules by heart, not necessarily understand them, and only be able to apply them in a limited environment. The advanced driver has driven a car many time, and in many different situations: in nice weather, in the rain, and maybe even on an icy surface. The advanced driver has driven different cars, and perhaps even driven in places where they drive on the "wrong" side of the road. Maybe the advanced driver would not be able to pass the theoretical driver's license test again, but he or she drives with a deep understanding of that foundation every day.

The ISTQB software testing advanced level certification is in fact three different certifications. You can choose to become:

- ▶ Advanced Level Test Management Professional;
- ▶ Advanced Level Test Analyst Professional;
- ▶ Advanced Level Technical Test Analyst Professional.



The syllabus covers all aspects of the three different certification paths and explains in great detail what you need to know and be able to do for each of them. The syllabus also explains how the examination is conducted. An advanced level certification does not come easy. You have to:



- ▶ Listen to the experiences and opinions of other testing professionals,
- ▶ Read as many of the books from the syllabus reference list as you can manage
- ▶ Think about what you have heard and seen and compare it to your own experiences: what have I seen, what was similar and what was different, and how and why and with what effect
- ▶ Use what you learn in your daily work as much as possible
- ▶ Discuss what you hear, read, think, experience, and write with your colleagues and your boss; maybe try to get a mentor
- ▶ Write things down. When we put pen to paper, things take another form in our brain, so when you have read and talked about a topic, write down what it means to you and how you may apply it. Use drawings, tables, and lists to get an overview. Just make it simple; you don't have to write a book.

In doing all this in any mixture, the theory you get from books and courses will be transformed into active knowledge and practice: You will become a truly advanced and, it is hoped, also a professional tester.

There are a number of lists of things that you will have to learn by heart. Isn't that a wonderful concept: Learn by heart! You don't have to memorize things; just see if it is possible for you to remember them. You should do it because the things you have to memorize are the very cornerstone of your profession: those things that should be closest to your professional heart.

### I.2.2 Ethics for Testers



A certain professional conduct is expected from professional people. This is also the case for testers. We can be placed in situations where we are faced with difficult choices and responsibilities.

The following code of ethics for testers is taken directly from the syllabus and should be known, understood, and followed by all professional testers.

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



## I.3 Software Testing Basics

In any profession it is a must that you have a firm understanding of the basic concepts of the profession. We will therefore briefly review the most important issues in testing to make sure that the fundamentals are present in our minds at all times.

### I.3.1 Terms and Definitions in Testing

There is no universal set of definitions of test concepts! That is a fact we have to live with, and part of being an advanced tester is the ability to map one set of definitions to others.

This book is based on the ISTQB Glossary of Terms used in Software Testing Version 1.2, April 2006 and the extension to the glossary included in the ISTQB Certified Tester Advanced Level Syllabus Version 2007 Beta.





### I.3.2 Testing Is Multidimensional

The universe of *testing is multidimensional*. It changes its composition and look constantly, depending on the circumstances. It is like looking into a kaleidoscope on a richly faceted picture.

Unfortunately, the different facets that comprise this universe can only be presented one at a time in a sequential way in a book. Even in a three-dimensional drawing, it would not be possible to capture the complexity of the testing universe.

It is your task and challenge as an advanced tester to grasp all the facets one by one and to be able to make different pictures of the testing universe depending on the situation you find yourself in at any given time. An unlimited number of different pictures of the testing universe may be made, and no two pictures will ever be exactly identical.

The testing universe facets include, but are not necessarily limited to, those listed here (in alphabetical order). Some of them might not mean anything to you at the moment, but at some point in time they all will.



- ▶ Coding languages
- ▶ Standards
- ▶ Development models
- ▶ Testing obstacles
- ▶ Development paradigms
- ▶ Testing progress
- ▶ Incidents
- ▶ Test approaches
- ▶ Incident handling
- ▶ Test basis
- ▶ Maturity models
- ▶ Test effort
- ▶ Money
- ▶ Test levels
- ▶ People skills
- ▶ Test objectives
- ▶ People types
- ▶ Test policy
- ▶ Process improvement
- ▶ Test processes
- ▶ Product architectures
- ▶ Test process improvement
- ▶ Product paradigms
- ▶ Test project risks
- ▶ Product risks
- ▶ Test scopes
- ▶ Quality assurance activities
- ▶ Test techniques
- ▶ Quality factors
- ▶ Test tools
- ▶ Quality goals
- ▶ Test types
- ▶ Resources
- ▶ Time
- ▶ Risk willingness

All the facets are discussed in this book, some in great detail, some just superficially, none to exhaustion.

Don't despair. Read and reread the chapters and sections in any order. Read other books. Try things out. Discuss with colleagues, both testing colleagues and others. Figure out what each facet means to you and how the facets can relate to each other in your world. Train and train again in making the picture that suits the situation you are in.



### I.3.3 Definition of Testing

So what is testing all about? Most of us have an idea of what testing is—something about finding errors. But further than that the confusion is fairly big.

Let's try to seek help in the standards.

IEEE 610 (Software Engineering Terminology): "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."



IEEE 829 (Test Documentation): "The process of analyzing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software items."

BS 7925-1 (Software Testing—Vocabulary): "Process of exercising software to verify that it satisfies requirements and to detect errors."

ISTQB Glossary of Terms used in Software Testing V 1.0: "The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects."

There is one term they all agree on: process. *Testing is a process*. So what does that process entail? IEEE 610 and BS 7925-1, respectively, talk about "operating" and "exercising"; that is, the idea that testing requires the software to run on a computer. This is what is also called "dynamic testing." IEEE 829 broadens the idea to "analyzing," thus including "static testing." And ISTQB takes the full step and includes both "dynamic and static." *Testing is both dynamic and static*.



Then what do we do dynamic and static testing on? The object of the testing in the definitions ranges from "system or component," "software item," and "software" to "software products and related work products." In line with testing being both dynamic and static, we have to conclude: *Testing can be done on any work product or product* (where the difference is that work products are not delivered to the customer).



And last but not least: Why? The reasons given include “observing,” “evaluate,” “detect the ... bugs/errors,” “to verify/determine ... satisfaction,” “to demonstrate ... fit for purpose.” We shall see later that all this boils down to: *Testing gathers information about the quality of the object under test.*



The quality is the amount of fulfillment of expectations. On one hand we have some expectations, and on the other hand, we have the product that should fulfill these expectations. The question is: Does it? We test to be able to answer that question.



Talking about quality, how does test relate to quality assurance? IEEE 610 defines: *“Quality assurance: A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.”*

Is there any difference between testing and quality assurance here? Not really. At least within the framework of this book, testing and quality assurance of the work products and the product will be considered as one and the same thing.



As Lee Copeland puts it:

*“Testing is comparing what is to what should be”*

and we could add:

*“and share the information obtained.”*

### I.3.3.1 Necessity of Testing



There should be no need to tell the readers of this book this, but we'll do it anyway: *It is necessary to test!*

Nobody is perfect,  
not even testers

*"Errare humanum est,"* (it is human to err), is imputed to a number of people. One source quotes a certain Hieronymus (app. 345–419). Cicero is quoted to have said: *"Errare humanum est, ignoscere divinum,"* "To err is human, to forgive divine" (*Philippicae orationes*).

Another quote without a source is: *"Cuiusvis hominis est errare, nullius nisi insipientis in errore perseverare,"* "Anybody can err, but only the fool persists in his defect."

It seems to be a recognized condition of life that we are not perfect and hence happen to make mistakes.

### Mistakes are not made on purpose!



Human beings are not machines that perform their tasks mechanically step by step. There are a number of life conditions that cause us to err.

Most people are able to handle  $7 \pm 2$  issues at one time. When this limit is passed, we forget things or mix information. We also tend to neglect or postpone issues that seem to be too difficult for us to handle. Sometimes we believe or hope that if we close our eyes to a problem it will somehow go away.

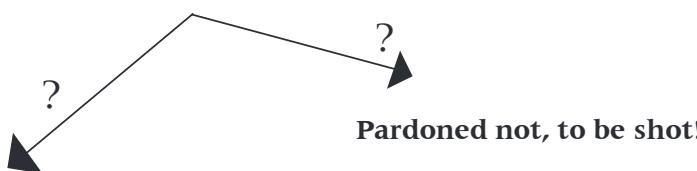
In our daily work we are distracted and disturbed numerous times. Streams of thoughts are broken, and important information is unintentionally left out.

People also seem to have a tendency to get used to things, which in the beginning seems wrong. Little by little, we internalize it and make the mistake ourselves; this it is called the "adaptive testing syndrome," but it also exists outside testing. For example, this is one of the reasons why our languages change over time.

Sometimes we don't express ourselves clearly, and that can lead to very dangerous guesswork if we don't go back to the source and ask for clarification. Just consider this requirement:

### **Pardon not to be shot!**

What does it mean?



**Pardon, not to be shot!**

Wrong assumptions, whether they are conscious or not, may also cause mistakes, and here the worst ones are the unconscious assumptions, so be very aware of those.

### Ex.

An accounting system was once implemented by a small software house. After a while it appeared that some invoices had identical numbers. It turned out that the developer didn't know that invoice numbers were not to be reused, even if an invoice had been "deleted" or archived. The accountant who had written the specification had not mentioned this issue, because "I thought it was common knowledge."

#### I.3.3.2 Handling Failures, Defects, and Mistakes

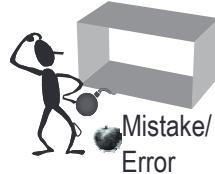
In a professional software development context, it is not precise enough to talk about errors as indiscriminately as we do in everyday language.

We therefore operate with three different terms: mistake (or error), defect (or defect), and failure, as illustrated here.



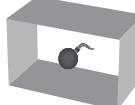
#### Mistake:

Human action that produces an incorrect result.



#### Defect:

Manifestation of an error in software.



#### Failure:

Deviation of the software from its expected delivery or service.



What happens is that a mistake, made by a human being, causes a defect to be placed in a product, for example, in a software module. The defect causes no harm as long as it is not encountered by anybody, but if it is "hit" during the use of the product, it will give rise to a failure.

Remember that the product—the test object—can be anything from the first requirements specification to the final product to be delivered to the customer. *It is important to distinguish between the concepts of "mistake," "defect," and "failure."*

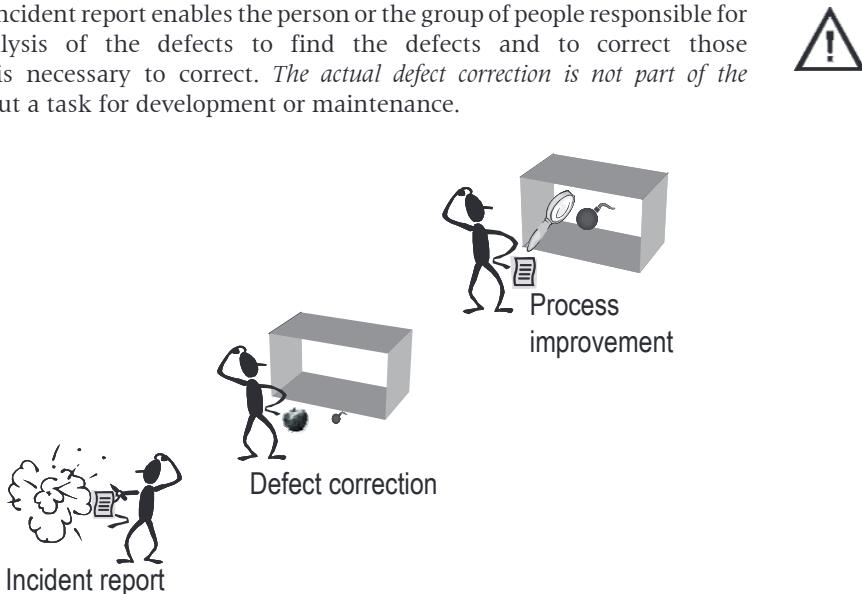
This is because they appear for different reasons, as can be seen above, and because they are to be treated very differently in the organization.



The job of the tester is to provoke as many failures as possible before the product reaches the customer.

When the tester sees a failure, he or she must fill out an incident report, describing what happened, and give this report to whoever is responsible for deciding what is then going to happen.

An incident report enables the person or the group of people responsible for the analysis of the defects to find the defects and to correct those that it is necessary to correct. *The actual defect correction is not part of the testing*, but a task for development or maintenance.



Process improvement uses analysis of incident reports to find areas where mistakes may be prevented by new ways of working (new processes) or caught earlier by better quality assurance processes.

A tester is testing the discount calculation in a sales support system. She enters the item she wants to “buy” and the number of units she wants. The system shows the price for one unit and calculates the total price. In the case where the tester “buys” 9 units, the price that the system shows is too high compared to what the tester has calculated beforehand and hence expects. The tester notes that she has seen a deviation. It turns out that the system calculates a discount when the number of units is equal to or greater than 9. But the requirement states that a discount shall be calculated if the number of units is 10 or more. There is a defect in the statement that determines if a discount shall be calculated or not. It further turns out that the designer happened to get it wrong when he wrote the detailed design for the requirement and that this was not discovered during the review of the design.

Ex.

## Questions

1. Which lists of terms are used in this book?
2. What are the chapters of the book?
3. What is the difference between a foundation and an advanced certification?
4. What is it you need to do when you study?
5. What are the eight areas for ethics for testers?
6. Which facets have an influence on testing?
7. How does ISTQB define testing?
8. How does Lee Copeland define testing?
9. Why is testing necessary?
10. What could be causes for mistakes?
11. What is the difference between “mistake,” “defect,” and “failure”?
12. What activity is used to reduce the likelihood of mistakes in software development?

## Appendix IA Vignettes

The vignettes used in the margins of this book are shown here to make it easier to refer to them when you are reading the book.



Caution



Definition



Example



Important



Overview



Reference



Remember



# CHAPTER

# 1

## Contents

- 1.1 Testing in the Software Life Cycle
- 1.2 Product Paradigms
- 1.3 Metrics and Measurement

## Basic Aspects of Software Testing

Testing is not an isolated activity, nor is it a development activity. Testing is a support activity: meaningless without the development processes and not producing anything in its own right: nothing developed entails nothing to test.

Testing is, however, a very important part of the life cycle of any product from the initial idea, during development, and in deployment until the product is taken out of deployment and disposed of.

Testing has its place intertwined with all these activities. Testing must find its place and fill it as well as possible.

### 1.1 Testing in the Software Life Cycle

The intention of product development is to somehow go from the vision of a product to the final product.



To do this a development project is usually established and carried out. The time from the initial idea for a product until it is delivered is the development life cycle.

When the product is delivered, its real life begins. The product is in use or deployed until it is disposed of. The time from the initial idea for a product until it is disposed of is called the product life cycle, or software life cycle, if we focus on software products.

Testing is a necessary process in the development project, and testing is also necessary during deployment, both as an ongoing monitoring of how the product is behaving and in the case of maintenance (defect correction and possibly evolution of the product).

Testing fits into any development model and interfaces with all the other development processes, such as requirements definition and coding. Testing also interfaces with the processes we call supporting processes, such as, for example, project management.

Testing in a development life cycle is broken down into a number of test levels—for example component testing and system testing. Each test level has its own characteristics.

### 1.1.1 Development Models



Everything we do in life seems to follow a few common steps, namely: conceive, design, implement, and test (and possibly subsequent correction and retest).

The same activities are recognized in software development, though there are normally called:

- Requirements engineering;
- Design;
- Coding;
- Testing (possibly with retesting, and regression testing).



Building blocks

In software development we call the building blocks “stages,” “steps,” “phases,” “levels,” or “processes.”

The way the development processes are structured is the development life cycle or the development model. A life cycle model is a specification of the order of the processes and the transition criteria for progressing from one process to the next, that is, completion criteria for the current process and entry criteria for the next.

Software development models provide guidance on the order in which the major processes in a project should be carried out, and define the conditions for progressing to the next process. Many software projects have experienced problems because they pursued their development without proper regard for the process and transition criteria.

The reason for using a software development model is to produce better quality software faster. That goal is equal for all models. *Using any model is better than not using a model.*

A number of software development models have been deployed throughout the industry over the years. They are usually grouped according to one of the following concepts:

- ▶ Sequential;
- ▶ Iterative;
- ▶ Incremental.



The building blocks—the processes—are the same; it is only a matter of their length and the frequency with which they are repeated.

The sequential model is characterized by including no repetition other than perhaps feedback to the preceding phase. This makeup is used in order to avoid expensive rework.



#### 1.1.1.1 Sequential Models

The *assumptions* for sequential models are:

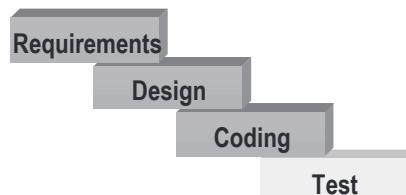


- ▶ The customer knows what he or she wants.
- ▶ The requirements are frozen (changes are exceptions).
- ▶ Phase reviews are used as control and feedback points.

The characteristics of a successful sequential development project are:

- ▶ Stable requirements;
- ▶ Stable environments;
- ▶ Focus on the big picture;
- ▶ One, monolithic delivery.

Historically the first type of sequential model was *the waterfall model*. A pure waterfall model consists of the building blocks ordered in one sequence with testing in the end.

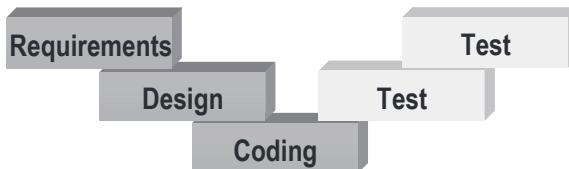




The goals of the waterfall model are achieved by enforcing fully elaborated documents as phase completion criteria and formal approval of these (signatures) as entry criteria for the next.



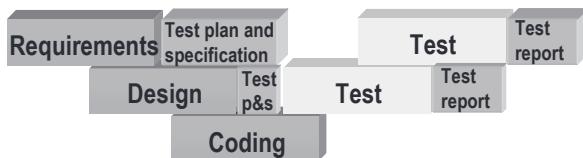
The V-model is an expansion of the pure waterfall model introducing more test levels, and the concept of testing not only being performed at the end of the development life cycle, even though it looks like it.



The V-model describes a course where the left side of the V reflects the processes to be performed in order to produce the pieces that make up the physical product, for example, the software code. The processes on the right side of the V are test levels to ensure that we get what we have specified as the product is assembled.

The pure V-model may lead you to believe that you develop first (the left side) and then test (the right side), but that is not how it is supposed to work.

A *W-model* has been developed to show that the test work, that is, the production of testing work products, starts as soon as the basis for the testing has been produced. Testing includes early planning and specification and test execution when the objects to test are ready. The idea in the V-model and the W-model is the same; they are just drawn differently.



When working like this, we describe what the product must do and how (in the requirements and the design), and at the same time we describe how we are going to test it (the test plan and the specification). This means that we are starting our testing at the earliest possible time.

The planning and specification of the test against the requirements should, for example, start as soon as the requirements have reached a reasonable state.

A W-model-like development model provides a number of advantages:



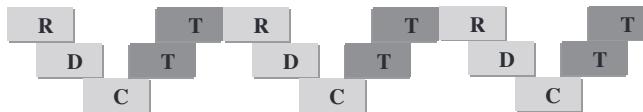
- ▶ More time to plan and specify the test
- ▶ Extra test-related review of documents and code
- ▶ More time to set up the test environment(s)
- ▶ Better chance of being ready for test execution as soon as something is ready to test

For some classes of software (e.g., safety critical systems, or fixed-price contracts), a W-model is the most appropriate.

### 1.1.1.2 Iterative and Incremental Models

In iterative and incremental models the strategy is that frequent changes should and will happen during development. To cater for this the basic processes are repeated in shorter circles, iterations. These models can be seen as a number of mini W-models; testing is and must be incorporated in every iteration within the development life cycle.

This is how we could illustrate an iterative or incremental development model.



The goals of an iterative model are achieved through various prototypes or subproducts. These are developed and validated in the iterations. At the end of each iteration an operational (sub)product is produced, and hence the product is expanding in each iteration. The direction of the evolution of the product is determined by the experiences with each (sub)product.

Note that the difference between the two model types discussed here is:

- ▶ In *iterative development* the product is not released to the customer until all the planned iterations have been completed.
- ▶ In *incremental development* a (sub)product is released to the customer after each iteration.



The *assumptions* for an iterative and incremental model are:



- ▶ The customer cannot express exactly what he or she wants.
- ▶ The requirements will change.
- ▶ Reviews are done continuously for control and feedback.

The characteristics of a successful project following such a model are:



- ▶ Fast and continuous customer feedback;
- ▶ Floating targets for the product;
- ▶ Focus on the most important features;
- ▶ Frequent releases.

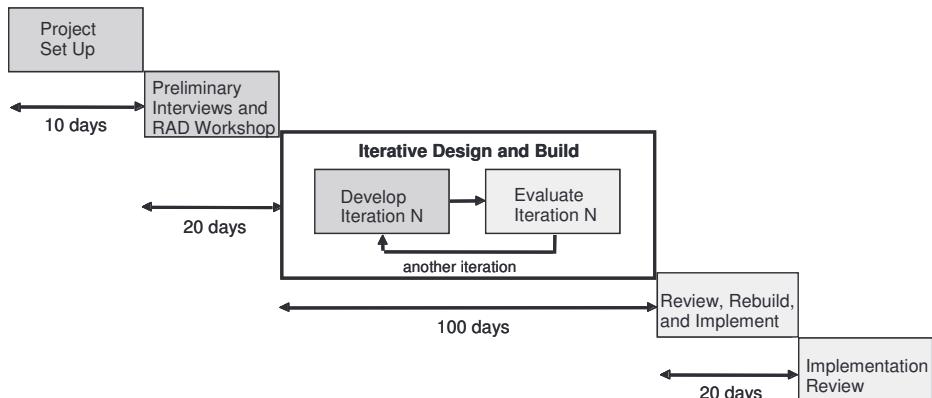
The iterative/incremental model matches situations in which the customers say: "I can't tell you what I want, but I'll know it when I see it"—the last part of the sentence often expressed as "IKIWISI."

These models are suited for a class of applications where there is a close and direct contact with the end user, and where requirements can only be established through actual operational experience.

A number of more specific iterative models are defined. Among these the most commonly used are the RAD model and the Spiral model.

The *RAD model* (Rapid Application Development) is named so because it is driven by the need for rapid reactions to changes in the market. James Martin, consultant and author, called the "guru of the information age", was the first to define this model. Since then the term RAD has more or less become a generic term for many different types of iterative models.

The original RAD model is based on development in timeboxes in few—usually three—iterations on the basis of fundamental understanding of the goal achieved before the iterations start. Each iteration basically follows a waterfall model.



When the last iteration is finished, the product is finalized and implemented as a proper working product to be delivered to the customer.

Barry Boehm, TRW Professor of Software Engineering at University of Southern California, has defined a so-called *Spiral Model*. This model aims at accommodating both the waterfall and the iterative model. The model consists of a set of full cycles of development, which successively refines the knowledge about the future product. Each cycle is risk driven and uses prototypes and simulations to evaluate alternatives and resolve risks while producing work products. Each cycle concludes with reviews and approvals of fully elaborated documents before the next cycle is initiated.

The last cycle, when all risks have been uncovered and the requirements, product design, and detailed design approved, consists of a conventional waterfall development of the product.

In recent years a number of incremental models, called *evolutionary* or *agile development models*, have appeared. In these models the emphasis is placed on values and principles, as described in the “Manifesto of Software Development.” These are:

- ▶ Individuals and interactions are valued over processes and tools
- ▶ Working software is valued over comprehensive documentation
- ▶ Customer collaboration is valued over contract negotiation
- ▶ Responding to change is valued over following a plan



One popular example of these models is the eXtreme Programming model, (XP). In XP one of the principles is that the tests for the product are developed first; the development is test-driven.

The development is carried out in a loosely structured small-team style. The objective is to get small teams (3–8 persons) to work together to build products quickly while still allowing individual programmers and teams freedom to evolve their designs and operate nearly autonomously.

These small teams evolve features and whole products incrementally while introducing new concepts and technologies along the way. However, because developers are free to innovate as they go along, they must synchronize frequently so product components all work together.

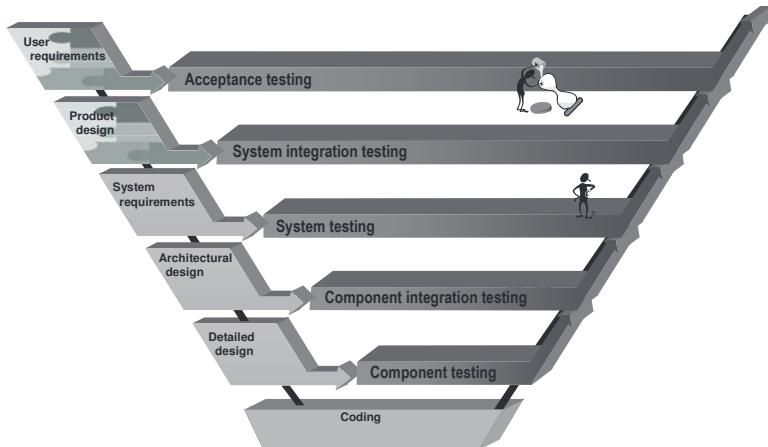


Testing is perhaps even more important in iterative and incremental development than in sequential development. The product is constantly evolved and extensive regression testing of what has previously been agreed and accepted is imperative in every iteration.



### 1.1.2 Dynamic Test Levels

In the V-model, and hence in basically all development, each development process has a corresponding dynamic test level as shown here.



The V-model used here includes the following dynamic test levels:

- ▶ Acceptance testing—based on and testing the fulfillment of the user requirements;
- ▶ System testing—based on and testing the fulfillment of the (software) system requirements;
- ▶ Component integration testing—based on and testing the implementation of the architectural design;
- ▶ Component testing—based on and testing the implementation of the detailed design.



Note that coding does not have a corresponding test level; it is not a specification phase, where expectations are expressed, but actual manufacturing! The code becomes the test object in the dynamic test levels.



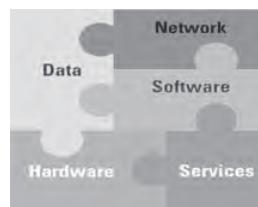
There is no standard V-model. The V-model is a principle, not a fixed model. Each organization will have to define its own so that it fits with the nature of the products and the organization. The models can have different make-ups, that is there may be more or less specification phases on the left side and hence testing levels on the right side, and/or the phases and levels may be named differently in different organizations.

In cases where the final product is part of or in itself a complex product it is necessary to consider more integration test levels.

In the case of a system of systems development project, described in Section 1.2.1, we will need a system integration test level.

Sometimes the product we are developing consists of a number of different types of systems, like for example:

- ▶ Software
- ▶ Hardware
- ▶ Network
- ▶ Data
- ▶ Services



In such cases there will be product design specification phases to distribute the requirements on the different systems in the beginning of the development life cycle and we will therefore need more or more integration test levels, such as, for example, hardware-software system integration and software-data system integration.

*Note:* the puzzle does NOT indicate possible interfaces between systems, only the fact that a product may be made up of different types of systems.

We could also be producing a product that is going to interface with system(s) the customer already has running. This will require a customer product integration test level.

No matter how many test levels we have, each test level is different from the others, especially in terms of goals and scope.

The organizational management must provide test strategies specific to each of the levels for the project types in the organization in which the testing is anchored. The contents of a test level strategy are discussed in Section 3.2.2.

Based on this the test responsible must produce test plans specific for each test level for a specific project. The contents of a test plan are discussed in Section 3.2.3.

The specific test plans for the test levels for a specific project should outline the differences between the test levels based on the goals and scope for each.

The fundamental test process is applicable for all the test levels. The test process is described in detail in Chapter 2.

The dynamic test levels in the V-model used here are discussed next.

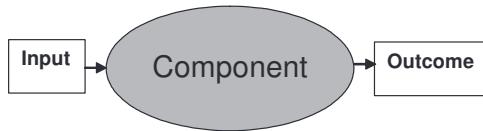
### 1.1.2.1 Component Testing

Component testing is the last test level where work, that is, planning, can start, the first where test execution can start, and therefore also the first to be finished.



The *goal* is to find defects in the implementation of each component according to the detailed design of the component.

The test object is hence individual components in isolation, and the basis documentation is the detailed design, and sometimes also other documentation like the requirements specification.



*It is not always easy to agree on what a component is.* A component could be what is contained in a compilable file, a subroutine, a class, or ... the possibilities are legion. The important thing in an organization is to define “a component”—it is less important what a component is defined as.

The scope for one component test is the individual component and the full scope of the component testing could be all components specified in the design, though sometimes only the most critical components may be selected for component testing.

An overall *component test plan* should be produced specifying the order in which the testing of the components is to take place. If this is done sufficiently early in the development, we as testers may be able to influence the development order to get the most critical components ready to test first. We also need to consider the subsequent component integration testing, and plan for components with critical interfaces to be tested first. For each component a very short plan (who, when, where, and completion criteria) and a test specification should be produced.

The assignment of the responsibility for the component testing depends on the level of independence we need. The lowest level of independence is where the manufacturer—here the developer—tests his or her own product. This often happens in component testing. The next level of independence is where a colleague tests his or her colleague’s product. This is advisable for component testing. The level of independence to use is guided by the risk related to the product. Risk management is discussed in Section 3.5.

The *techniques* to use in component testing are functional (black-box) techniques and structural (white-box) techniques. Most often tests are first designed using functional techniques. The coverage is measured and more tests can be designed using structural techniques if the coverage is not sufficient to meet the completion criteria.



*The code must never be used as the basis documentation from which to derive the expected results.*



Nonfunctional requirements or characteristics, such as memory usage,

defect handling, and maintainability may also be tested at the component testing level.

To isolate a component, it is necessary to have a driver to be able to execute the component. It is also usually necessary to have a stub or a simulator to mimic or simulate other components that interface with the component under test. Test stubs are sometimes referred to as test harness.

The needs for test drivers and stubs must be specified as part of the specification of the test environment. Any needed driver and stubs must, of course, be ready before each individual component testing can start.

Many tools support component testing. Some are language-specific and can act as drivers and stubs to facilitate the isolation of the component under test. Tools are discussed in Chapter 9.

Component *test execution* should start when the component has been deemed ready by the developer, that is when it fulfils the entry criteria. The least we require before the test execution can start is that the component can compile. It could be a very good idea to require that a static test and/or static analysis has been performed and approved on the code as entry criteria for the component test execution.

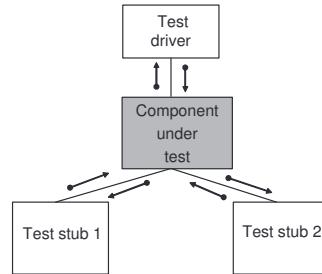
Measures of time spent on the testing activities, on defects found and corrected, and on obtained coverage should be collected. This is sometimes difficult because component testing is often performed as an integrated development/testing/debugging activity with no registration of defects and very little if any reporting. This is a shame because it deprives the organization of valuable information about which kinds of defects are found and hence the possibility for introducing relevant process improvement.

The component testing for each individual component must stop when the completion criteria specified in the plan have been met. For each component a very short summary report should be produced.

A summary *report* for the collection of components being tested should be produced when the testing has been completed for the last component.

Any *test procedures* should be kept, because they can be very useful for later confirmation testing and regression testing. Drivers and stubs should be kept for the same reason, and because they can be useful during integration testing as well.

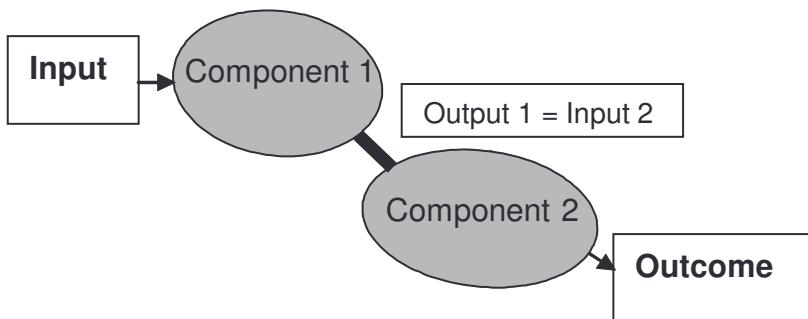
The goals of integration testing are to find defects in the interfaces and invariants between interacting entities that interact in a system or a product. Invariants are substates that should be unchanged by the interaction between two entities.



### 1.1.2.2 Integration Testing

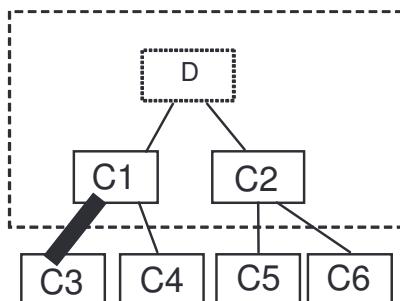
The objective is not to find defects inside the entities being integrated—the assumption being that these have already been found during previous testing.

The entities to integrate may be components as defined in the architectural design or different systems as defined in the product design. The principles for integration testing are the same no matter what we are integrating.



For the collection of interfaces to test an overall integration test plan should be produced specifying, among other things, the order in which this testing is to take place. There are four different strategies for the testing order in integration testing:

- ▶ Top down;
- ▶ Bottom up;
- ▶ Functional integration;
- ▶ Big-bang.



In top-down integration the interfaces in the top layer in the design hierarchy are tested first, followed by each layer going downwards. The main program serves as the driver.

This way we quickly get a “shell” created. The drawback is that we (often) need a large number of stubs.

In bottom-up integration the interfaces in the lowest level are tested first. Here higher components are replaced with drivers, so we may need many drivers. This integration strategy enables early integration with hardware, where this is relevant.

In functional integration we integrate by functionality area; this is a sort of vertically divided top-down strategy. We quickly get the possibility of having functional areas available.

In big-bang integration we integrate most or everything in one go. At first glance it seems like this strategy reduces the test effort, but it does not—on the contrary. It is impossible to get proper coverage when testing the interfaces in a big-bang integration, and it is very difficult to find any defects in the interfaces, like looking for a needle in a haystack. Both top-down and bottom-up integration often end up as big-bang, even if this was not the initial intention.

For each interface a *very short plan* (who, when, where, and completion criteria) and a test specification should be produced. Often one of the producers of the entities to integrate has the responsibility for that integration testing, though both should be present.

Both the formality and the level of independence is higher for system integration testing than for component integration, but these issues should not be ignored for component integration testing.

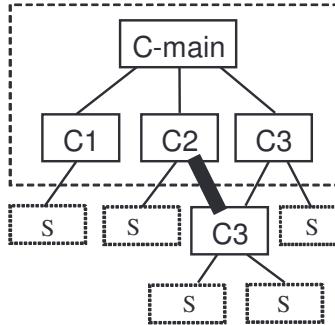
The *techniques* to use must primarily be selected among the structural techniques, depending on the completion criteria defined in the plan. Non-functional requirements or characteristics, such as performance, may also be tested at the integration testing level.

The necessary drivers or stubs must be specified as part of the environment and developed before the integration testing can start. Often stubs from a previous test level, for example, component testing, can be reused.

The *execution* of the integration testing follows the completion of the testing of the entities to integrate. As soon as two interacting entities are tested, their integration test can be executed. There is no need to wait for all entities to be tested individually before the integration test execution can begin.

Measures of time spent on the testing, on defects found and corrected, and on coverage should be collected.

The integration testing for each individual interface must stop when the completion criteria specified in the plan have been met. A very short test report should be produced for each interface being tested. We must keep on integrating and testing the interfaces and the invariants until all the entities



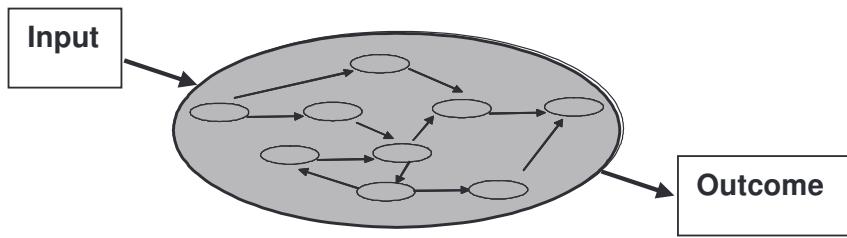
have been integrated and the overall completion criteria defined in the integration test plan have been met.

A summary report for the collection of interfaces being tested should be produced when the testing has been completed for the last interface.

### 1.1.2.3 System Testing



The goal of system testing is to find defects in features of the system compared to the way it has been defined in the software system requirements. The test object is the fully integrated system.



The better the component testing and the component integration testing has been performed prior to the system testing, the more effective is the system testing. All too often system testing is impeded by poor or missing component and component integration testing.

A comprehensive system test plan and system test specification must be produced.

The system test specification is based on the system requirements specification. This is where all the expectations, both the functional and the non-functional should be expressed. The functional requirements express what the system shall be able to do—the functionality of the system. The non-functional requirements express how the functionality presents itself and behaves. In principle the system testing of the two types of requirements is identical. We test to get information about the fulfillment of the requirements.

The techniques to use will most often be selected among the functional techniques, possibly supplemented with experience-based techniques (exploratory testing, for example), depending on the completion criteria defined in the plan. *Experience-based test techniques should never be the only techniques used in the system testing.*



The execution of system test follows the completion of the entire component integration testing. It is a good idea to also require that a static test

has been performed on the requirements specification and on the system test specification before execution starts.

Many tools support system testing. Capture/replay tools and test management tools are especially useful to support the system testing.

Measures of time spent on the testing, on faults found and corrected, and on coverage should be collected. The system testing must stop when the completion criteria specified in the plan have been met.

A system test report should be produced when the system testing has been completed.

#### 1.1.2.4 Acceptance Testing

The acceptance testing is the queen's inspection of the guard. The goal of this test level is not, like for all the other ones, to find defects by getting the product to fail. At the acceptance test level the product is expected to be working and it is presented for acceptance.

*The customer and/or end users must be involved* in the acceptance testing. In some cases they have the full responsibility for this testing; in other cases they just witness the performance.

In the acceptance testing the test object is the entire product. That could include:

- ▶ Business processes in connection with the new system;
- ▶ Manual operations;
- ▶ Forms, reports, and so forth;
- ▶ Document flow;
- ▶ Use cases and/or scenarios.



The techniques are usually mostly experience-based, where the future users apply their domain knowledge and (hopefully) testing skills to the validation of the product. Extracts of the system test specification are sometimes used as part of the acceptance test specification.

An extra benefit of having representatives of the users involved in the acceptance testing is that it gives these users a detailed understanding of the new system—it can help create ambassadors for the product when it is brought into production.

There may be a number of acceptance test types, namely:

- ▶ Contract acceptance test;
- ▶ Alpha test;
- ▶ Beta test.



The contract acceptance test may also be called factory acceptance test. This test must be completed before the product may leave the supplier; the product has to be accepted by the customer. It requires that clear acceptance criteria have been defined in the contract. A thorough registration of the results is necessary as evidence of what the customer acceptance is based on.

An alpha test is usage of the product by representative users at the development site, but reflecting what the real usage will be like. Developers must not be present, but extended support must be provided. The alpha test is not used particularly often since it can be very expensive to establish a “real” environment. The benefits rarely match the cost.

A beta test is usage of the product by selected (or voluntary) customers at the customer site. The product is used as it will be in production. The actual conditions determine the contents of the test. Also here extended support of the users is necessary. Beta tests preferably run over a longer period of time. Beta tests are much used for off-the-shelf products—the customers get the product early (and possibly cheaper) in return for accepting a certain amount of immaturity and the responsibility for reporting all incidents.

### 1.1.3 Supporting Processes

No matter how the development model is structured there will always be a number of supporting activities, or supporting processes, for the development.

The primary supporting processes are:

- Quality assurance;
- Project management;
- Configuration management.

These processes are performed during the entire course of the development and support the development from idea to product.

Other supporting processes may be:

- Technical writing (i.e., production of technical documentation);
- Technical support (i.e., support of environment including tools).

The supporting processes all interface with the test process.

*Testing is a product quality assurance activity* and hence actually part of the supporting processes. This is in line with the fact that testing is meaningless without the development processes and not producing anything in its own right: nothing developed entails nothing to test.

The test material, however, is itself subject to quality assurance or testing, so testing is recursive and interfaces with itself. Testing also interfaces with



project management and configuration management as discussed in detail in the following.

Testing also interfaces with technical writing. The documentation being written is an integrated part of the final product to be delivered and must therefore also be subject to quality assurance (i.e., to static testing).

When the product—or an increment—is deployed, it transfers to the maintenance phase. In this phase corrections and possibly new features will be delivered at defined intervals, and testing plays an important part here.



### 1.1.3.1 Product Quality Assurance

It is not possible to test quality into a product when the development is close to being finished. The quality assurance activities must start early and become an integrated part of the entire development project and the mindset of all stakeholders.

Quality assurance comprises four activities:

- ▶ Definition of quality criteria
- ▶ Validation
- ▶ Verification
- ▶ Quality reporting

*Note* that the validation is not necessarily performed before the verification; in many organizations it is the other way around, or in parallel.



First of all, the *Quality criteria* must be defined. These criteria are the expression of the quality level that must be reached or an expression of “what is sufficiently good.” These criteria can be very different from product to product. They depend on the business needs and the product type. Different quality criteria will be set for a product that will just be thrown away when it is not working than for a product that is expected to work for many years with a great risk of serious consequences if it does not work.

There are two quality assurance activities for checking if the quality criteria have been met by the object under testing, namely:

- ▶ Validation;
- ▶ Verification.

They have different goals and different techniques. The object to test is delivered for validation and verification from the applicable development process.



*Validation* is the assessment of the correctness of the product (the object) in relation to the users' needs and requirements.

You could also say that validation answers the question: "*Are we building the correct product?*"

Validation must determine if the customer's needs and requirements are correctly captured and correctly expressed and understood. We must also determine if what is delivered reflects these needs and requirements.

When the requirements have been agreed upon and approved, we must ensure that during the entire development life cycle:

- Nothing has been forgotten.
- Nothing has been added.

It is obvious that if something is forgotten, the correct product has not been delivered. It does, however, happen all too often, that requirements are overlooked somewhere in the development process. This costs money, time, and credibility.

On the surface it is perhaps not so bad if something has been added. But it does cost money and affect the project plan, when a developer—probably in all goodwill—adds some functionality, which he or she imagines would be a benefit for the end user.



What is worse is that *the extra functionality will probably never be tested* in the system and acceptance test, simply because the testers don't know anything about its existence. This means that the product is sent out to the customers with some untested functionality and this will lie as a mine under the surface of the product. Maybe it will never be hit, or maybe it will be hit, and in that case the consequences are unforeseeable.

The possibility that the extra functionality will never be hit is, however, rather high, since the end user will probably not know about it anyway.

Validation during the development process is performed by analysis of trace information. If requirements are traced to design and code it is an easy task to find out if some requirements are not fulfilled, or if some design or code is not based on requirements.

The ultimate validation is the user acceptance test, where the users test that the original requirements are implemented and that the product fulfills its purpose.



*Verification*, the other quality assurance activity, is the assessment of whether the object fulfills the specified requirements.

Verification answers the question: "*Are we building the product correctly?*"

The difference between validation and verification can be illustrated like this:



Validation confirms that a required calculation of discount has been designed and coded in the product.

Verification confirms that the implemented algorithm calculates the discount as it is supposed to in all details.

A number of techniques exist for verification. The ones to choose depend on the test object.

In the early phases the test object is usually a document, for example in the form of:

- ▶ Plans;
- ▶ Requirements specification;
- ▶ Design;
- ▶ Test specifications;
- ▶ Code.

The verification techniques for these are the static test techniques discussed in Chapter 6:



- ▶ Inspection;
- ▶ Review (informal, peer, technical, management);
- ▶ Walk-through.

Once some code has been produced, we can use static analysis on the code as a verification technique. This is not executing the code, but verifying that it is written according to coding standards and that it does not have obvious data flow faults. Finally, dynamic testing where the test object is executable software can be used.

We can also use dynamic analysis, especially during component testing. This technique reveals faults that are otherwise very difficult to identify. Dynamic analysis is described in Section 4.6.



A little memory hint:



**Validation**

**Correct**

a comes before e  
t comes before y

**Verification**

**Correctly**

*Quality assurance reports* on the findings and results should be produced.

If the test object is not found to live up to the quality criteria, the object is returned to development for correction. At the same time incident reports should be filled in and given to the right authority.

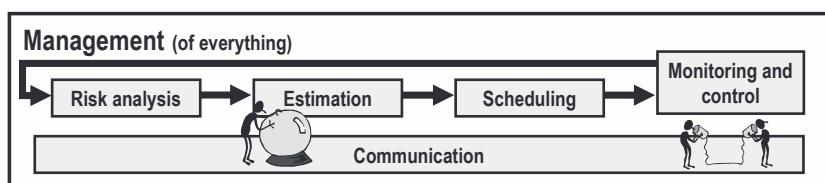
Once the test object has passed the validation and verification, it should be placed under configuration management.

### 1.1.3.2 Project Management

It is obviously important that the development process and the supporting processes are managed and controlled during the entire project. Project management is the supporting process that takes care of this, from the first idea to the release.

The most important activities in project management are:

- ▶ Risk analysis;
- ▶ Estimation;
- ▶ Scheduling;
- ▶ Monitoring and control;
- ▶ Communication.



Test management is subordinated to project management.

The estimation, risk analysis, and scheduling of the test activities will either have to be done in cooperation with the project management or by the test manager and consolidated with the overall project planning. The results of the monitoring and control of the test activities will also have to be coordinated with the project management activities.

The project management activities will not be discussed further here.

The corresponding test management activities are described in detail in Chapter 3.



### 1.1.3.3 Configuration Management

Configuration management is another supporting process with which testing interacts. The purpose of configuration management is to establish and maintain integrity of work products and product.

Configuration management can be defined as:

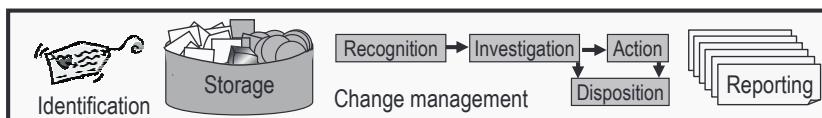
- ▶ Unique identification;
- ▶ Controlled storage;
- ▶ Change management (recognition, investigation, action, and disposition);
- ▶ Status reporting.

for selected

- ▶ Work products;
- ▶ Product components;
- ▶ Products.

during the entire life time of the product.

An object under configuration management is called a configuration item.



The purpose of *identification* is to uniquely identify each configuration item and to specify its relations to the outside world and to other configuration items. Identification is one of the cornerstones of configuration management, as it is impossible to control something for which you don't know the identity.

Each organization must define the conventions for unique identification of the configuration items.

 Ex.**Test cases**

10.3.1.6 (80) Test for correct bank identity number 1.A

The identification encompasses:

Current section number in document: 10.3.1.6

Running unique number: 80

Version of test case: 1.A

The purpose of *storage* is to ensure that configuration items don't disappear or are damaged. It must also be possible to find the items at any time and have them delivered in the condition in which we expect to find them.

Storage is something physical. Items that are stored are physically present at a specific place. This place is often called the library, or the controlled library.

Configuration items are released from storage to be used as the basis for further work. *Usage* is all imaginable deployment of configuration items without these being changed, not just usage of the final product by the final users.

Usage may for instance be a review, if a document is placed under configuration management in the form of a draft and subsequently has to be reviewed.

It may be testing of larger or minor parts of the system, integration of a subcomponent into a larger component, or proper operation or sale of a finished product.

Configuration items released from storage must not be changed, ever! But new versions may be issued as the result of change control.

The purpose of *change management* or *change control* is to be fully in control of all change requests for a product and of all implemented changes. Any change should be traced to the configuration item where the change has been implemented.

The initiation of change control is the occurrence of an incident. Incident management is discussed in Chapter 7.

The purpose of *status reporting* is to make available the information necessary for effective management of the development, testing, and maintenance of a product, in a useful and readable way.

Configuration management can be a well of information.

A few words about the concept of a *configuration item* are needed here. In principle everything may be placed under configuration management. The following list shows what objects may become configuration items, with the emphasis on the test ware.

- ▶ Test material: Test specifications, test data(base), drivers, and stubs
- ▶ Environments: Operating systems, tools, compilers, and linkers

- Technical documentation: Requirements, design, and technical notes
- Code: Source code, header files, include files and system libraries
- Project documentation: User manuals, build scripts, data, event registrations, installation procedures, and plans
- Administrative documents: Letters, contracts, process description, sales material, templates, and standards
- Hardware: Cables, mainframe, PC, workstation, network, storage, and peripherals



#### 1.1.3.4 Technical Writing

Technical writing is a support process much used in the United Kingdom. Other European countries do not use technical writers that much—here the developers, testers, and the rest of the project team are left to their own devices.

Technical writers are people with special writing skills and education. They assist other staff members when difficult issues need to be made clear to the intended audience in writing.

We as testers interface with technical writers in two ways:

- We subject their work to static tests.
- We use their work in our testing.

We can of course also use their skills as writers, but that does not happen very often. Testers usually write for other testers and for a technical audience.

## 1.2 Product Paradigms

The use of computers to assist people in performing tasks has developed dramatically since the first huge (in physical size) computers were invented around the middle of the last century. The first computers were about the size of a family home and you could only interact with them via punch cards or tape and printed output. Those were the days.

Today we as testers may have to cope with a number of different product types or product paradigms, and with different development paradigms and coding languages. Not all of us encounter all of them, but it is worth knowing a little bit about the challenges they each pose for us.

We always need to be aware of the product and development paradigm used for the (testing) projects we are involved in. We must tailor our test approach and detailed test processes to the circumstances and be prepared to tackle any specific obstacles caused by these as early as possible.

A few significant product paradigms are discussed here.



### 1.2.1 Systems of Systems

A system of systems is the concept of integrating existing systems into a single information system with only limited new development. The concept was first formulated by the American Admiral Owens in his book *Lifting the Fog of War*. The concept was primarily used for military systems but is spreading more and more to civilian systems as well.

The idea is to use modern network technologies to exploit the information spread out in a number of systems by combining and analyzing it and using the results in the same or other systems to make these even more powerful.

**Ex.**

A tiny example of a system of systems is a sprinkling system at a golf course. The gardener can set the sprinkling rate for a week at the time. Using a network connection this system is linked to a system at the meteorological institute where hours of sunshine, average temperatures, and rainfall are collected. This information is sent to a small new system, which calculates the needed sprinkling rate on a daily basis and feeds this into the sprinkling system automatically. The gardener's time, water, the occasional flooding, and the occasional drying out of the green is saved.

Systems of systems are complicated in nature. The final system is usually large and complex as each of the individual systems may be. Each of the individual systems may in itself consist of a number of different subsystems, such as software, hardware, network, documentation, data, data repository systems, license agreements, services (e.g., courses and upgrades), and descriptions of manual processes. Few modern systems are pure software products, though they do exist.

Even if the individual systems are not developed from scratch these systems pose high demands on supporting the supporting processes, especially project management, but also configuration management and product quality assurance. In the cases where some or all of the individual systems are being developed as part of the construction of a system of systems this poses even higher demands in terms of communication and coordination.

From a testing point of view, there are at least three important aspects to take into account when working with systems of systems:



- ▶ System testing of the individual systems
- ▶ Integration testing of systems
- ▶ Regression testing of systems and integration

A system of systems is only as strong as the weakest link, and the completion criteria for the system testing of each individual system must reflect the quality expectations toward the complete system of systems. The system testing of each of the individual systems is either performed as part of the project, or assurance of its performance must be produced, for example in the form of test reports from the producer.

Systems of systems vary significantly in complexity and may be designed in hierarchies of different depths, ranging from a two-layer system where the final system of systems is composed of a number of systems of the same “rank” to many-layered (system of (systems of (systems of systems))). Integration of the systems must be planned and executed according to the overall architecture, observing the integration testing aspects discussed in Section 1.1.2.

It is inevitable that defects will be found during system and integration testing of systems of systems, and significant iterations of defect correction, confirmation testing, and not least regression testing must be anticipated and planned for. *Strict defect handling is necessary to keep this from getting out of control*, resulting, for example, in endless correction and recorrection circles.

Systems of systems may well contain systems of types where special care and considerations need to be made for testing. Examples may be:

- ▶ Safety-critical systems
- ▶ Large mainframe systems
- ▶ Client-server systems
- ▶ Web-based systems
- ▶ PC-based systems
- ▶ Embedded systems
- ▶ Real-time systems
- ▶ Object-oriented development



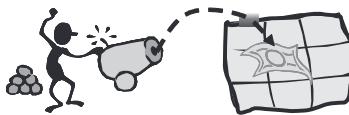
### 1.2.2 Safety-Critical Systems

Any system presents some risk to its owners, users, and environment. Some present more than others, and those that present the most risk are what we call safety-critical systems.

The risk is a threat to something valuable. All systems either have something of value, which may be jeopardized, inside them, or their usage may jeopardize some value outside them. A system should be built to protect the values both from the result of ordinary use of the system and from the result of malicious attacks of various kinds.

A typical categorization of values looks at values concerning:

- ▶ Safety
- ▶ Economy
- ▶ Security
- ▶ Environment



Many regulatory standards address how to determine the safety criticality of systems and provide guidelines for the corresponding testing. Some of them (but probably not all) are:



- ▶ CEI/IEC 61508—Functional safety of electrical/electronic/programmable safety-related systems
- ▶ DO-178-B—Software considerations in airborne systems and equipment certification
- ▶ pr EN 50128—Software for railway control and protection systems
- ▶ Def Stan 00-55—Requirements for safety-related software in defense equipment
- ▶ IEC 880—Software for computers in the safety systems of nuclear power stations
- ▶ MISRA—Development guidelines for vehicle-based software
- ▶ FDA—American Food and Drug Association (pharmaceutical standards)



The standards are application-specific, and that can make it difficult to determine what to do if we have to do with multidisciplinary products. Nonetheless, standards do provide useful guidance. The most generic of the standards listed above is IEC 61508; this may always be used if a system does not fit into any of the other types.

All the standards operate with so-called software integrity levels (SILs).

This table shows an example of a classification.

**Ex.**

SIL Value \ SIL	A (100.000.000)	B (100.000)	C (100)	D (1)
<b>Safety</b>	Many people killed	Human lives in danger	Damage to physical objects; risk of personal injury	Insignificant damage to things; no risk to people
<b>Economy</b>	Financial catastrophe (the company must close)	Great financial loss (the company is threatened)	Significant financial loss (the company is affected)	Insignificant financial loss
<b>Security</b>	Destruction/disclosure of strategic data and services	Destruction/ disclosure of critical data and services	Faults in data	No risk for data
<b>Environment</b>	Comprehensive and irreparable damage to the environment	Reparable, but comprehensive damage to the environment	Local damage to the environment	No environmental risk

The concept of SILs allows a standard to define a hierarchy of levels of testing (and development). A SIL is normally applied to a subsystem; that is, we can operate with various degrees of SILs within a single system, or within a system of systems. The determination of the SIL for a system under testing is based on a risk analysis.

The standards concerning safety critical systems deal with both development processes and supporting processes, that is, project management, configuration management, and product quality assurance.

We take as an example the CEI/IEC 61508 recommends the usage of test case design techniques depending on the SIL of a system. This standard defines four integrity levels: SIL4, SIL3, SIL 2, and SIL1, where SIL4 is the most critical.

**Ex.**

For a SIL4-classified system, the standard says that the use of equivalence partitioning is highly recommended as part of the functional testing. Furthermore the use of boundary value analysis is highly recommended, while the use of cause-effect graph and error guessing are only recommended. For white-box testing the level of coverage is highly recommended, though the standard does not say which level of which coverage.

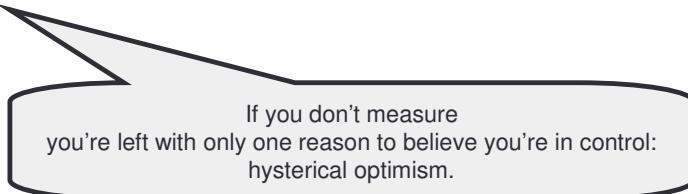
The recommendations are less and less strict as we come down the SILs in the standard.



For highly safety-critical systems the testers may be required to deliver a compliance statement or matrix, explaining how the pertaining regulations have been followed and fulfilled.

### 1.3 Metrics and Measurement

Tom De Marco, one of the testing gurus, once said:



If you don't measure  
you're left with only one reason to believe you're in control:  
hysterical optimism.

One of the principles of good planning, both of testing and anything else, is to define specific and measurable goals for the activities. But it is not enough for goals to be measurable; we must also collect facts that can tell us if we have indeed achieved the goals.

#### 1.3.1 Measuring in General

For facts or data collection we operate with the following concepts:

- Metric—A definition of what to measure, including data type, scale, and unit
- Measuring method—The description of how we are going to get the data
- Measurements—The actual values collected for the metrics



An example could be that the metric for the size of a book is “number of pages”; the measuring method is to “look at the last page number”; and the measurement for *Alice in Wonderland*, ISBN 7409746, is “54.”

It is a good idea to establish a measurement plan as part of the project plan or master test plan. This should specify the metrics we want to measure and the measuring methods, who is going to measure, and perhaps most importantly: how the measurements will be analyzed and used.

Our measurements are derived from raw data such as time sheets, incident reports, test logs, and work sheets. *Direct measurements* are measurements we get directly from the raw data, for example, by counting the number of log sheets for passed test procedures and counting the number of incident reports. *Indirect measurements* are measurements we can calculate from direct measurements.

Most direct measurements have no meaning unless they are placed in relation to something. Number of incidents as such—for example, 50—says nothing about the product or the processes. But if we calculate the defects found compared to the estimated amount of defects it gives a much better indication—either of our estimation or of the quality of the product!



It is a common mistake to think that only objective data should be used. Objective data is what you can measure independently of human opinions. But even though subjective data has an element of uncertainty about it, it can be very valuable. Often subjective data is even cheaper to collect than objective data.

A subjective metric could be:

The opinion of the participants in walk-throughs concerning the usefulness of the walk-through activity on a scale from 1 to 5, where 1 is lowest and 5 is highest.



This is easy to collect and handle, and it gives a good indication of the perception of the usefulness of walk-throughs.

The metrics should be specified based on the goals we have set and other questions we would like to get answers to, such as how far we are performing a specific task in relation to the plan and expectations.

### 1.3.2 Test-Related Metrics

Many, many measurements can be collected during the performance of the test procedures (and any other process for that matter). They can be divided into groups according to the possibilities for control they provide. The groups and a few examples of direct measurements are listed here for inspirational purposes; the lists are by no means exhaustive.



#### ► Measurements about progress

- Of test planning and monitoring:
  - Tasks commenced
  - Task completed
- Of test development:
  - Number of specified and approved test procedures
  - Relevant coverages achieved in the specification, for example, for code structures, requirements, risks, business processes
  - Other tasks completed
- Of test execution and reporting:
  - Number of executed test procedures (or initiated test procedures)
  - Number of passed test procedures

- Number of passed confirmation tests
- Number of test procedures run as regression testing
- Other tasks completed
- Of test closure:
  - Tasks completed

For each of these groups we can collect measurements for:

- Time spent on specific tasks both in actual working hours and elapsed time
- Cost both from time spent and from direct cost, such as license fees

**‣ Measurements about coverage:**

- Number of coverage elements covered by the executed test procedures code structures covered by the test

**‣ Measurements about incidents:**

- Number of reported incidents
- Number of incidents of different classes, for example, faults, misunderstandings, and enhancement requests
- Number of defects reported to have been corrected
- Number of closed incident reports

**‣ Measurements about confidence:**

- Subjective statements about confidence from different stakeholders



All these measurements should be taken at various points in time, and the time of the measuring should be noted to enable follow-up on the development of topics over time, for example the development in the number of open incident reports on a weekly basis.

It is equally important to prepare to be able to measure and report status and progress of tasks and other topics in relation to milestones defined in the development model we are following.

To be able to see the development of topics in relation to expectations, corresponding to factual and/or estimated total numbers are also needed. A few examples are:

- Total number of defined test procedures
- Total number of coverage elements
- Total number of failures and defects
- Actual test object attributes, for example, size and complexity
- Planned duration and effort for tasks
- Planned cost of performing tasks

### 1.3.3 Analysis and Presentation of Measurements

It is never enough to just collect measurements. They must be presented and analyzed to be of real value to us. The analysis and presentation of measurements are discussed in Section 3.4.2.



### 1.3.4 Planning Measuring

It is important that stakeholders agree to the definition of the metrics and measuring methods, before any measurements are collected. Unpopular or adverse measurements may cause friction, especially if these basic definitions are not clear and approved. You can obtain very weird behaviors by introducing measurements!

There is some advice you should keep in mind when you plan the data you are going to collect. You need to aim for:



- **Agreed metrics**—Definitions (for example, what is a line of code), scale (for example, is 1 highest or lowest), and units (for example, seconds or hours) must be agreed on and understood
- **Needed measures**—What is it you want to know, to monitor, and to control?
- **Repeatable measurements**—Same time of measure and same instrument must give the same measurement
- **Precise measurements**—Valid scale and known source must be used
- **Comparable measurements**—For example, over time or between sources
- **Economical measurements**—Practical to collect and analyse compared to the value of the analysis results
- **Creating confidentiality**—Never use measurements to punish or award individuals
- **Using already existing measurements**—Maybe the measurements just need to be analyzed in a new way
- **Having a measurement plan**—The plan should outline what, by whom, when, how, why
- **Using the measurements**—Only measure what can be used immediately and give quick and precise feedback



## Questions

1. What is the development life cycle and the product life cycle?
2. What are the building blocks in software development models?
3. What are the basic development model types?

4. What is the difference between a waterfall development model and a V-model?
5. What are the advantages of a W-model?
6. What is the main difference between iterative development and incremental development?
7. What are the characteristics of projects following an iterative model?
8. What does RAD stand for?
9. What is the principle in Boehm's spiral model?
10. What are the value principles for agile development?
11. What is the most interesting aspect of XP from a testing point of view?
12. What is the standard V-model that everybody should follow?
13. What is the test object in component testing?
14. What are stubs and drivers used for?
15. When should an individual component test stop?
16. What are the test objects in integration testing?
17. Which integration strategies exist?
18. Which techniques could be used in system testing?
19. How is acceptance testing different from the other test levels?
20. What are the supporting processes discussed in this book?
21. What are the four quality-assurance activities?
22. What is validation?
23. Why is gold-plating dangerous?
24. What is verification?
25. What are the five project management activities?
26. What are the four configuration management activities?
27. What are the purposes of each of them?
28. What can be placed under configuration management from a testing point of view?
29. What is the interface between testing and technical writers?
30. What is a system of systems?
31. What should be considered when testing a system of systems?
32. What are the value categories in safety critical systems?
33. What is a SIL?
34. How can standards guide the testing of a safety-critical system?
35. What is the difference between a direct and an indirect measurement?
36. Why can subjective measurements be useful?
37. What are the groups for which testers can collect information?
38. When should measuring take place?
39. How should measurements be used?
40. What is the most important aspect of measurements apart from using them?

## CHAPTER

# 2

## Testing Processes

Everything we do, from cooking a meal to producing the most complicated software products, follows a process. A process is a series of activities performed to fulfill a purpose and produce a tangible output based on a given input.

The process view on software development is gaining more and more interest. Process models are defined to assist organizations in process improvement—that is, in making their work more structured and efficient.

Testing can also be regarded as a process.

Like all processes the test process can be viewed at different levels of detail. An activity in a process can be seen as a process in its own right and described as such. The generic test process consists of five activities or processes. Each of these is treated like a separate and complete process.

Test development is what is usually understood as the real test work. This is sometimes divided into two subprocesses, namely:

- ▶ Test analysis and design;
- ▶ Test implementation and execution.

The borderline between the two subprocesses is blurred and the activities are iterative across this borderline.

The two subprocesses are, however, discussed individually here.

### Contents

- 2.1 Processes in General
- 2.2 Test Planning and Control
- 2.3 Test Analysis and Design
- 2.4 Test Implementation and Execution
- 2.5 Evaluating Exit Criteria and Reporting
- 2.6 Test Closure

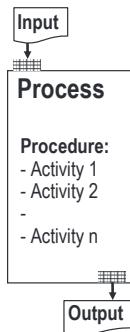
## 2.1 Processes in General

### 2.1.1 The Concept of a Process

A process is a series of activities performed to fulfill a specific purpose. Based on an input to the process and following the activities—also called the procedure—a tangible output is produced.

It is important to remember that the tangible output (for example, a specification) is not the goal itself. The goal is to perform the activities, to think, to discuss, to try things out, to make decisions, to document, and whatever else is needed. The tangible output is the way of communicating how the purpose of the process has been fulfilled.

Processes can be described and hence monitored and improved. A process description must always include:



- ▶ A definition of the input
- ▶ A list of activities—the procedure
- ▶ A description of the output

In the basic description of a process, the purpose is implicitly described in the list of activities.

For a more comprehensive and more useful process description the following information could also be included:

- ▶ Entry criteria—What must be in place before we can start?
- ▶ Purpose—A description of what must be achieved ?
- ▶ Role—Who is going to perform the activities?
- ▶ Methods, techniques, tools—How exactly are we going to perform the activities?
- ▶ Measurements—What metrics are we going to collect for the process?
- ▶ Templates—What should the output look like?
- ▶ Verification points—Are we on the right track?
- ▶ Exit criteria—What do we need to fulfill before we can say that we have finished?

A process description must be operational. It is not supposed to fill pages and pages. It should fit on a single page, maybe even a Web page, with references to more detailed descriptions of methods, techniques, and templates.

### 2.1.2 Monitoring Processes

It is the responsibility of management in charge of a specific area to know how the pertaining processes are performed. For testing processes it is of course important for the test leader to know how the testing is performed and progressing.

Furthermore, it is important for the people in charge of process improvement to be able to pinpoint which processes should be the target processes for improvement activities and to be able to predict and later determine the effect of process improvement activities.

This is why the description for each process should include the metrics we are interested in for the process, and hence the measurements we are going to collect as the process is being performed.

Metrics and measurements were discussed in Section 1.3, and Section 3.4 discusses how test progress monitoring and control can be performed. In this chapter a few metrics associated with the activities in each of the test processes are listed for inspiration.



### 2.1.3 Processes Depend on Each Other

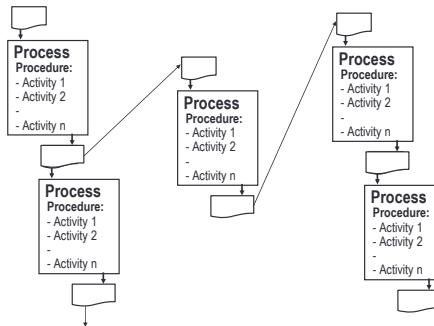
The input to a process must be the output from one or more proceeding process(es)—except perhaps for the very first, where the infamous napkin with the original idea is the input. The output from a process must be the input to one or more other processes—even the final product, which is the input to the maintenance process.

The dependencies between processes can be depicted in a process model, where it is shown how outputs from processes serve as inputs to other processes.

A process model could be in a textual form or it could be graphical, as shown in the following figure. Here, for example, the output from the top-left process serves as input to the top-middle process and to the lower-left process.



Processes depend on  
each other.  
Output  $n$  = Input  $m$



The figure only shows a tiny extract of a process model, so some of the processes deliver input to processes that are not included in the figure.

### 2.1.4 The Overall Generic Test Process

Testing is a process. The generic test process defined in the ISTQB foundation syllabus can be described like this:

The purpose of the test process is to provide information to assure the quality of the product, decisions, and the processes for a testing assignment.



The inputs on which this process is based are:

- ▶ Test strategy
- ▶ Project plan
- ▶ Master test plan
- ▶ Information about how the testing is progressing



The activities are:

- ▶ Test planning and control
- ▶ Test development
- ▶ Test analysis and design
- ▶ Test implementation and execution
- ▶ Evaluating exit criteria and reporting
- ▶ Test closure activities



The output consists of:

- ▶ Level test plan
- ▶ Test specification in the form of test conditions, test design, test cases, and test procedures and/or test scripts
- ▶ Test environment design and specification and actual test environment including test data
- ▶ Test logs
- ▶ Progress reports
- ▶ Test summary report
- ▶ Test experience report



The generic test process is applicable for each of the dynamic test levels to be included in the course of the development and maintenance of a product. So the process should be used in testing such as:



- ▶ Component testing
- ▶ Integration testing
- ▶ System testing
- ▶ Acceptance testing



The test levels are described in Chapter 1.

Since we apply the view that the concept of testing covers all types of product quality assurance, the generic test process is also applicable to static test (reviews), static analysis (automated static test), and dynamic analysis (run-time analysis of programs). Static testing is described in Chapter 6.

The static test type processes and the level specific test processes depend on each other; and each of them hook into other development processes and support processes. This is described in Chapter 1.

An example of process dependencies is:

The software requirements specification—output from the software requirements specification process—is input to an inspection process and to the system test process.

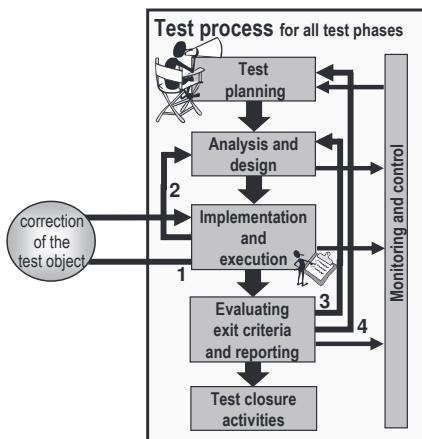


There are many more dependencies. Some of them are described in the following sections.



*The test activities need not be performed in strict sequential order.* Test planning and control are constant activities in the sense that they are not just done once in the beginning of the test assignment. Monitoring of the process should be done on an ongoing basis, and controlling and replanning activities performed when the need arises. Sometimes test analysis and design is performed in parallel with test implementation and execution. A model is not a scientific truth; when using a model, even a very well-defined model, we should be open for necessary tailoring to specific situations.

*The generic test process is iterative—not a simple straightforward process.* It must be foreseen that we'll have to perform the activities more than once in an iterative way before the exit criteria have been fulfilled. The iterations to be foreseen in the test process are shown in the figure here.



Experience shows that in most cases three iterations must be reckoned with as a minimum before the test process can be completed.



The first activity from which an iteration may occur is the test implementation and execution. This is where we detect the failures, when the actual result is different from the expected.

The resulting iterations may be:

- 1 The defect is in the test object.



A calculation does not give the expected result, and it appears that the algorithm for the calculation has been coded wrongly.

When the defect has been corrected we must retest the software using the test procedure that encountered the failure in the first place. We'll probably also perform some regression testing.

- 2 The defect is in the test procedure.



A calculation does not give the expected result, but here it appears that the test case was wrong.

The defect must be corrected and the new test case(s) must be executed. This iteration usually goes back to the analysis and design activity.

The second activity from which an iteration may occur is the evaluation of the exit criteria. This is where we find out if the exit criteria are not met.

The resulting iteration in this situation may be:

- 3 More test cases must be specified to increase coverage, and these must then be executed.



In the checking it turns out that the decision coverage for a component is only 87%. One more test case is designed and when this is executed the coverage reaches 96%.

- 4 The exit criteria are relaxed or strengthened in the test plan.



The coverage is found to be too small because of an-error handling routine that is very hard to reach. The required coverage for the component is relaxed to 85%.

The generic test process described in detail in the following is primarily aimed for a scripted test where the test is specified before the execution starts.



This does not mean that this test process is not useful for other techniques. Even in exploratory testing where you test a little bit and direct the further test based on the experience gained, you need to plan and control what is going on, to analyze and design (at least in your head), to execute, to report (very important!), and to close the testing.

### 2.1.5 Other Testing Processes

The test process defined in the ISTQB syllabus is just one example of a possible testing process. Each organization should create its own test process suitable for the specific circumstances in the organization.

A test process may be created from scratch or it may be created as a tailoring of a standard process.

The various process improvement models that exist provide frameworks for processes. Some cover all the process areas in a development organization; others cover the test area in details. Some of the most used process improvement models are discussed in Chapter 8. Two of these are presented here as examples of the framework such process models can provide.

One of the test specific models, *Test Process Improvement Model* (TPI), defines a list of 20 key areas. These cover the total test process and each of them is a potential process in its own right.

The 20 key areas are grouped into four so-called cornerstones as follows:

- ▶ Life cycle—Test strategy, life cycle model, moment of involvement
- ▶ Techniques—Estimating and planning, test specification techniques, static test techniques, metrics
- ▶ Infrastructure—Test tools, test environment, office environment
- ▶ Organization—Commitment and motivation, test functions and training, scope of methodology, communication, reporting, defect management, testware management, test process management, evaluating, low-level testing



The TPI model provides inspiration as to which activities could and should be specified for each of these areas when they are being defined as processes in an organization.

Another process model is the Critical Testing Processes (CTP). This model also defines a number of process areas. In this model the process areas are grouped into four classes:

- ▶ Plan—Establish context, analyze risks, estimate, plan
- ▶ Prepare—Grow and train team, create testware, test environment, and test processes
- ▶ Perform—Receive test object(s), execute and log tests
- ▶ Perfect—Report bugs, report test results, manage changes



## 2.2 Test Planning and Control

The purpose of the test planning process is to verify the mission of the testing, to define the objectives of the testing, and to make the necessary decisions to transform the test strategy into an operational plan for the performance of the actual testing task at hand.

The planning must first be done at the overall level resulting in a master test plan. The detailed planning for each test level is based on this master test plan. Identical planning principles apply for the overall planning and the detailed planning.

The purpose of the control part is to ensure that the planned activities are on track by monitoring what is going on and take corrective actions as appropriate.



The inputs on which this process is based are:

- ▶ Test strategy
- ▶ Master test plan
- ▶ Information about how the testing is progressing



The activities are:

- ▶ Verify the mission and define the objectives of the testing
- ▶ Decide and document how the general test strategy and the project test plan apply to the specific test level: what, how, where, who
- ▶ Make decisions and initiate corrective actions as appropriate as the testing progresses



The output consists of:

- ▶ Level test plan

### 2.2.1 Input to Test Planning and Control

The planning of a test level is based on the relevant test strategy, the project plan for the project to which the test assignment belongs, and the master test plan. The contents of these documents, as well as the detailed contents of the level test plan are discussed in Chapter 3.

The level test plan outlines how the strategy is being implemented in the specific test level in the specific project at hand. Basically we can say that the stricter the strategy is and the higher the risk is, the more specific must the level test plan be. Testing and risk is also discussed in Chapter 3.

The test level plan must be consistent with the master test plan. It must also be consolidated with the overall plan for the project in which the testing is a part. This is to ensure that schedules and resources correspond, and that other teams, which interface with the test team in question, are identified.

The decisions to make in the test planning and control process are guided by the expected contents of the test plan. Don't get it wrong: *The decisions are not made for the purpose of writing the plan, but for the purpose of getting agreement and commitment of all the stakeholders in the test to be performed.*

The planning and control of the test are continuous activities. The initial planning will take place first. Information from monitoring what is going on as the testing progresses may cause controlling actions to be taken. These ac-



tions will usually involve new planning and necessary corrections to be made in the plan when it no longer reflects the reality.

### 2.2.2 Documentation of Test Planning and Control

The tangible output of this process is the level test plan for the testing level to which the process is applied. The structure of the level test plan should be tailored to the organization.

In order not to start from scratch every time it is, however, a good idea to have a template. A template could be based on the IEEE 829 standard. This standard suggests the following contents of a test plan—the words in brackets are the corresponding concepts as defined in this syllabus:

Test plan identifier

1. Introduction (scope, risks, and objectives)
2. Test item(s) (test object(s))
3. Features to be tested
4. Features not to be tested
5. Approach (targets, techniques, templates)
6. Item pass/fail criteria (exit criteria including coverage criteria)
7. Suspension criteria and resumption requirements
8. Test deliverables (work products)
9. Testing tasks (analysis, design, implementation, execution, evaluation, reporting, and closure; all broken down into more detailed activities in an appropriate work break down structure)
10. Environmental needs
11. Responsibilities
12. Staffing and training needs
13. Schedule
14. Risks and contingencies



Test plan approvals

The level test plan produced and maintained in this process is input to all the other detailed test processes. They all have the level test plan as their reference point for information and decisions.

### 2.2.3 Activities in Test Planning

It cannot be said too often: *Test planning should start as early as possible*. The initial detailed planning for each of the test levels can start as soon as the documentation on which the testing is based has reached a suitable draft level.

The planning of the acceptance testing can start as soon as a draft of the user requirements is available.



Early planning has a number of advantages. It provides, for example, time to do a proper planning job, adequate time to include the stakeholders, early visibility of potential problems, and means of influencing the development plan (e.g., to develop in a sequence that expedites testing).

The test planning activities must first of all aim at setting the scene for the testing assignment for the actors in accordance with the framework. The test planning for a test level must verify the mission and define the objectives—that is the goal or purpose, for the testing assignment. Based on this the more detailed planning can take place.

”

### 2.2.3.1 Defining Test Object and Test Basis

The object of the testing depends on the test level as described in Chapter 1. Whatever the test object is, the expectations we have for it, and therefore what we are going to test the fulfillment of, should be described in the test basis.

The test planning must identify the test basis and define what it is we are going to test in relation to this. This includes determination of the coverage to achieve for the appropriate coverage item(s). The expected coverage must be documented in the level test plan as (part of) the completion criteria. The coverage items depend on the test basis.

Examples of the most common test basis and corresponding coverage items are listed in the following table.

Ex.

Test level	Test basis	Coverage items
Component testing	<ul style="list-style-type: none"> <li>▶ Requirements</li> <li>▶ Detailed design</li> <li>▶ Code</li> </ul>	<ul style="list-style-type: none"> <li>▶ Statements</li> <li>▶ Decisions</li> <li>▶ Conditions</li> </ul>
Component integration testing	<ul style="list-style-type: none"> <li>▶ Architectural design</li> </ul>	<ul style="list-style-type: none"> <li>▶ Internal interfaces</li> <li>▶ Individual parameters</li> <li>▶ Invariants</li> </ul>
System testing	<ul style="list-style-type: none"> <li>▶ Software requirements specification</li> </ul>	<ul style="list-style-type: none"> <li>▶ Requirements:           <ul style="list-style-type: none"> <li>- functional</li> <li>- nonfunctional</li> </ul> </li> </ul>
System integration testing	<ul style="list-style-type: none"> <li>▶ Product design</li> </ul>	<ul style="list-style-type: none"> <li>▶ External interfaces</li> <li>▶ Individual parameters</li> <li>▶ Invariants</li> </ul>
Acceptance testing	<ul style="list-style-type: none"> <li>▶ User requirements specification</li> <li>▶ User manual</li> </ul>	<ul style="list-style-type: none"> <li>▶ Requirements expressed as           <ul style="list-style-type: none"> <li>- use cases</li> <li>- business scenarios</li> </ul> </li> </ul>

Static test	<ul style="list-style-type: none"> <li>▶ Documents the static test is based on</li> </ul>	<ul style="list-style-type: none"> <li>▶ Pages</li> <li>▶ Requirements</li> <li>▶ Test cases</li> </ul>
-------------	---	---

Standards, both internal and external to the organization, may also be used as the test basis.

### 2.2.3.2 Defining the Approach

The test approach must be based on the strategy for the test at hand. This section expands the approach and makes it operational.

The approach must at least cover:

- ▶ The test methods and test techniques to use
- ▶ The structure of the test specification to be produced and used
- ▶ The tools to be used
- ▶ The interface with configuration management
- ▶ Measurements to collect
- ▶ Important constraints, such as availability or “fixed” deadline



for the testing we are planning for.

First of all, the test object determines the *method*:

- ▶ If the test object is something that can be read or looked at, the method is static test—the specific choice of static test type(s) depends on the criticality of the object.
- ▶ If the test object is executable software, the method is dynamic test.



For each of the dynamic test types a number of *test case design techniques* may be used. The test case design techniques are discussed in detail in Chapter 4. The choice of test techniques is dependent on the test object, the risks to be mitigated, the knowledge of the software under testing, and the nature of the basis document(s). The higher the risk, the more specific should the recommendation for the test case design techniques to use be, and the more thorough should the recommended test case design techniques be.

The *structure of the test specification* must be outlined here. Test specifications may be structured in many ways—for example, according to the structure suggested in IEEE 829. This is described in Section 2.3.2.





The *usage of tools* must also be described in the approach. Tools are described in Chapter 10. *The strategy for the tool usage must be adhered to.*

The interface with *configuration management* covers:

- ▶ How to identify and store the configuration items we produce in the test process
- ▶ How to get the configuration items we need (for example, design specifications, source code, and requirements specifications)
- ▶ How to handle traceability
- ▶ How to register and handle incidents



A reference to descriptions in the configuration management system, should suffice here, but we are not always that lucky. If no descriptions exist we must make them—and share them with those responsible for configuration management.

The *measurements* to be collected are used for monitoring and control of the progress of the testing. We must outline what and how to measure in the approach. Measurements are discussed in detail in Sections 1.3 and 3.4.



### 2.2.3.3 Defining the Completion Criteria

The completion criteria are what we use to determine if we can stop the testing or if we have to go on to reach the objective of the testing.



The completion criteria are derived from the strategy and should be based on a risk analysis; the higher the risk, the stricter the completion criteria; the lower the risk the less demanding and specific the completion criteria.

It is important to decide up front which completion criteria should be fulfilled before the test may be stopped.



The completion criteria guide the specification of the test and the selection of test case design techniques. These techniques are exploited to provide the test cases that satisfy the completion criteria. Test case design techniques are discussed in detail in Chapters 4 and 5.

The most appropriate completion criteria vary from test level to test level. Completion criteria for the test may be specified as follows:

- ▶ Specified coverage has been achieved
- ▶ Specified number of failures found per test effort has been achieved
- ▶ No known serious faults
- ▶ The benefits of the system are bigger than known problems
- ▶ (The time has run out)

The last one is not an official completion criterion and should never be used as such; it is nonetheless often encountered in real life!

Coverage is a very often used measurement and completion criteria in testing. Test coverage is the degree, expressed as a percentage, to which the coverage items have been exercised by a test.

The above mentioned completion criteria may be combined and the completion criteria for a test be defined as a combination of more individual completion criteria.

Examples of combinations of completion criteria for each of the test levels may be:

- ▶ Component testing
  - ▶ 100% statement coverage
  - ▶ 95% decision coverage
  - ▶ No known faults
- ▶ Integration testing (both for components and systems)
  - ▶ 90% parameter coverage
  - ▶ 60% interface coverage
  - ▶ No known faults
- ▶ System testing
  - ▶ 90% requirement coverage
  - ▶ 100% equivalence class coverage for specific requirements
  - ▶ No known failures of criticality 1 or 2
  - ▶ Stable number of failures per test hour for more than 20 test hours
- ▶ Acceptance testing
  - ▶ 100% business procedure coverage
  - ▶ No known failures of criticality 1



Ex.

#### 2.2.3.4 Defining Work Products and Their Relationships

The number of deliverables, their characteristics, and estimates of their sizes must be defined, not least because this is used as input for the detailed estimation and scheduling of all the test activities, but also because the precision of what is going to be delivered sets stakeholders' expectations.

Typical deliveries or work products from a test level are:

- ▶ Level test plan(s)
- ▶ Test specification(s)
- ▶ Test environment(s)
- ▶ Test logs and journals
- ▶ Test reports

The level test plan is the plan being specified in this process.

The test specification is a collective term for the result of the test design and implementation activities. This is the most complicated of the work products. It is important that the *structure of the test specification* is outlined in the level test plan, so that its complexity is understood and taken into consideration when the effort is estimated, and also to guide the work in the subsequent activities.

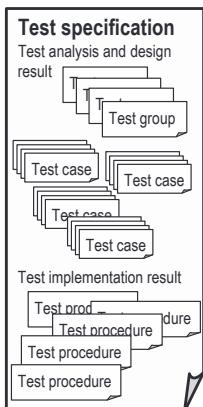
Test specifications may be structured in many ways. Each organization must figure out which structure is the most suitable for them. No matter the structure the test specification could be held in one document or in several separate documents; the physical distribution of the information is not important, but the actual contents are.

The structure shown and explained here is based on the structure suggested in IEEE 829. A full test specification may consists of:

- ▶ A *test design* consisting of a number of test groups (or designs) with test conditions and high-level test cases derived from the basis documentation. The designs will typically reflect the structure of the test basis documentation. The relationships between the elements in the basis documentation and the high-level test cases may well be quite complicated, often including even many-to-many relationships.
- ▶ A number of *low-level test cases* extracted from the high-level test cases and being made explicit with precise input and output specifications .
- ▶ A number of test procedures each encompassing a number of test cases to be executed in sequence. The relationships between high-level test cases and test procedures may also be complicated and include many-to-many relationships.

This structure is applicable to test specifications at all test levels, for example, for:

- ▶ Component testing
- ▶ Integration testing
- ▶ System testing
- ▶ Acceptance testing



The detailed contents of the test specification are discussed in Section 2.3.3.



### 2.2.3.5 Scoping the Test Effort

The definition of exhaustive testing is: test case design technique in which the test case suites comprise all combinations of input values and preconditions for component variables. No matter how much we as testers would like to do the ultimate good job, *exhaustive testing is a utopian goal*.

We do not have unlimited time and money; in fact we rarely have enough to obtain the quality of the testing we would like. It would in almost all cases take an enormous amount of resources in terms of time and money to test exhaustively and is therefore usually not worth it.

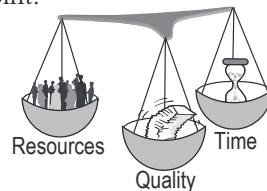
We have three mutually dependent parameters that we as testers need to balance. In fact, for everything we do, we need to balance these parameters, but here we'll look at them from a tester's view point.

The parameters are:

*Time*: The available calendar time

*Resources*: The available people and tools

*Quality*: The quality of the testing



These parameters form what we call the quality triangle, that is the triangle for the quality of work we can deliver.

In a particular project we need to initially achieve a balance between the time and resources spent on testing and the quality of the testing we want.

The basic principle of the quality triangle is: *It is not possible to change one of the parameters and leave the other two unchanged—and still be in balance!*

The time and the resources are fairly easy concepts to understand. Testing takes time and costs resources. The quality of the testing is more difficult to assess. The easiest way to measure that quality is to measure the test coverage. The test coverage is the percentage of what we set out to test (e.g., statements) that we have actually been able to cover with our test effort.

*Test coverage is a measure for the quality of the test.*

When we perform the test planning we need to look further ahead than the horizon of testing. Important factors could cause one of the parameters in the quality triangle for testing to be fixed.



It may, for example, be necessary:

- ▶ To fix a release date for economical or marketing reasons if the product must be presented at the yearly sales exhibition for the particular type of product
- ▶ To keep a given price, especially in fixed price projects
- ▶ To obtain a specific level of quality, for example in safety critical products





Everything needs to be balanced. The time and cost of testing to enhance the quality must be balanced with the cost of missing a deadline or having remaining defects in the product when it goes on the market.

### ***Work Breakdown Structure***

One of the things on which the test planning is based, is a list of all the tasks to be performed. This list should be in the form of a work breakdown structure of the test process at hand. If we use the test process defined here the overall tasks are planning, monitoring, control, analysis, design, implementation, execution, evaluation, reporting, and closure, all broken down into more detailed activities in an appropriate work breakdown structure.



The tasks, together with resources and responsibilities, are input items to the test schedule.



A list and a description of every single task must therefore be produced. If a task is not mentioned here it will probably not get done. *Be conscientious: remember to remember EVERYTHING!* Be as detailed as necessary to get a precise estimate. A rule of thumb is to aim at a break down of activities to tasks that can be done in no more than about 30 to 40 hours.

All the activities in the test process must be included in the task list. Do not forget to include the test management activities like planning, monitoring, and control. Also remember that the estimation and scheduling takes time—these activities must be included as well.

It is important here to remember that the test process is iterative. This must of course be taken into account during the estimation, but it will facilitate the estimation if iterations of activities are explicitly mentioned in the task list.

### ***Defining Test Roles***



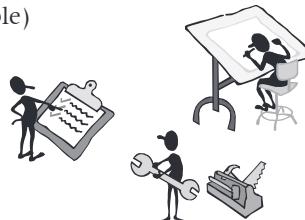
A (software test) project is like a play in which all roles must be filled in order for the play to be performed. Some roles are big, some are small, but they are all important for the whole.

Real people must fill the roles. Real people vary; they have different personalities, a fact of life that it is almost impossible to change. Technical skills you can learn, but your personality is to a large extent fixed when you reach adulthood. Different people fill different roles in different ways, and the differences between people may be used to the advantage of everybody, if the basics of team roles are known. This is discussed in detail in Chapter 10.

It is of great importance in the general understanding of the work to be done that the roles are described. Processes and procedures may be described thoroughly, but only when the activities and tasks are connected to roles and thereby to real people do they become really meaningful.

The roles to handle the testing tasks may be:

- ▶ Test leader (manager or responsible)
- ▶ Test analyst/designer
- ▶ Test executer
- ▶ Reviewer/inspector
- ▶ Domain expert
- ▶ Test environment responsible
- ▶ (Test)tool responsible



Test teams are formed by all these roles. We need different teams depending on which test phase we are working in, but the principles are the same:

- ▶ *All relevant roles must be present and filled in the team*
- ▶ A role can be filled by one person or more people, depending on the size of the testing assignment at hand
- ▶ One person can fill one role or more roles, again depending on the size (but keep in mind that less than 25% time for a role = 0% in real life)



The roles are assigned to organizational units and subsequently to named people. The necessary staff to fulfill the roles and take on the responsibilities must be determined.

The roles each require a number of specific skills. If these skills are not available in the people you have at your disposal, you must describe any training needs here. The training should then be part of the activities to put in the schedule.

### ***Producing the Schedule***

In scheduling the tasks, the staffing and the estimates are brought together and transformed into a schedule. Risk analysis may be used to prioritize the testing for the scheduling: the higher the risk, the more time for testing and the earlier the scheduled start of the testing task.

The result of this is a schedule that shows precisely who should do what at which point in time and for how long.

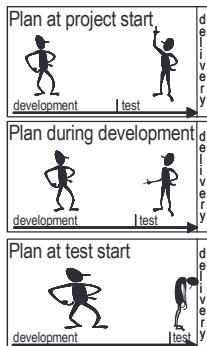
A framework for the resources and the schedule must be obtained from the overall project plan, and the result of the test scheduling must be reconciled with the project plan.

Estimations for all the tasks are input to the scheduling. Once the tasks are estimated they can be fitted into the project time line. Test estimation is discussed in detail in Section 3.3.



The schedule is also based on the actual people performing the tasks, the people's efficiency, and their availability.

### 2.2.4 Activities in Test Control



As the testing progresses the control part of test management is about staying in control and making necessary corrections to the plan when it no longer reflects the reality.

Measurements are collected in the test monitoring activities for all the detailed activities in the test processes, and these measurements are analyzed to understand and follow the actual progress of the planned test activities and the resulting coverage. Decisions must be made if things are deviating significantly from the plan, and corrective actions may be necessary.

The testing often gets pressed for time, since it is the last activity before the product is released. When development is delayed it is tempting to shorten the test to be able to keep the release date.

But if our testing time is cut, we have to change at least one other parameter in the quality triangle; anything else is impossible. It is important to point this out to management. It is irresponsible if for example the consequences on resources and/or testing quality of a time cut are not made clear. If it looks as if we are going to end up in the all too familiar situation illustrated here, we have to take precautions.

There is more about test monitoring and control in Section 3.4

”

### 2.2.5 Metrics for Test Planning and Control

Metrics to be defined for the monitoring and control of the test planning and control activities themselves may include:

- ▶ Number of tasks commenced over time
- ▶ Task completion percentage over time
- ▶ Number of tasks completed over time
- ▶ Time spent on each task over time

This will of course have to be compared to the estimates and schedule of the test planning and control activities.

## 2.3 Test Analysis and Design

The purpose of the test analysis and design activities is to produce test designs with test conditions and tests cases and the necessary test environment based on the test basis and the test goals and approach outlined in the test plan.

The inputs on which this process is based are:



- ▶ Level test plan
- ▶ Basis documentation



The activities are:

- ▶ Analysis of basis documentation
- ▶ Design of high-level test cases and test environment



The output consists of:

- ▶ Test design
- ▶ Test environment design and specification

### 2.3.1 Input to Test Analysis and Design

The input from the level test plan that we need for this process is:

- ▶ Test objectives
- ▶ Scheduling and staffing for the activities
- ▶ Definition of test object(s)
- ▶ Approach—especially test case design techniques to use and structure and contents of the test specification
- ▶ Completion criteria, not least required coverage
- ▶ Deliverables

We of course also need the test basis—that is, the material we are going to test the test object against.

### 2.3.2 Documentation of Test Analysis and Design

The result of the test analysis and design should be documented in the test specification. This document or series of documents encompasses

- ▶ The test designs—also called test groups
- ▶ The test cases—many test cases per test design
- ▶ Test procedures—often many-to-many relationship with test cases

The overall structure of the test specification is defined in the level test plan. The detailed structure is discussed below.

The test specification documentation is created to *document the decisions* made during the test development and to *facilitate the test execution*.



### 2.3.3 Activities in Test Analysis and Design

The idea in structured testing is that the test is specified before the execution. The test specification activity can already start when the basis documentation is under preparation.

The test specification aims at designing tests that provide the largest possible coverage to meet the coverage demands in the test plan. This is where test case design techniques are a great help.



The work with the specification of the test groups, the test conditions, the test cases, and the test procedures are highly iterative.



A side effect of the analysis is that we get an extra review of the basis documentation. Don't forget to feed the findings back through the correct channels, especially if the basis documentation isn't testable.

### 2.3.3.1 Defining Test Designs

In test design the testing task is broken into a number of test design or test groups. This makes the test development easier to cope with, especially for the higher test levels. Test groups may also be known as test topics or test areas.

A *test design* or test group specification should have the following contents according to IEEE 829:



Test design specification identifier

1. Features to be tested (test conditions)
2. Approach refinement
3. List of high-level test cases
4. List of expected test procedures
5. Feature pass/fail criteria

Test design specification approvals

The groups and the procedures must be uniquely identified. The number of test groups we can define depends on the test level and the nature, size, and architecture of the test object:



- ▶ In component testing we usually have one test group per component
- ▶ For integration testing there are usually a few groups per interface
- ▶ For system and acceptance testing we typically have many test groups



A few examples of useful test groups defined for a system test are:

- ▶ Start and stop of the system
- ▶ Functionality x
- ▶ Nonfunctional attribute xx
- ▶ Error situations



It should be noted that it is not very common to document the test design as thoroughly as described here. Often a list of groups with a short purpose description and list of the test procedures for each are sufficient.



#### **Test group: 2 (2) Handling member information**

The purpose of this test group is to test that the member information can be created and maintained.

Test procedure: 2.1 (10) Creating new member

Test procedure: 2.2 (14) Changing personal information

Test procedure: 2.3 (11) Changing bonus point information

Test procedure: 2.4 (13) Deleting member

The unique identification is the number in brackets, for example (10). The number before the unique identifier is the sorting order to ensure that the groups and procedures are presented in a logical order independently of the unique number, for example 2.1. The “disorder” of the unique identification is a sign of the iterative way in which they have been designed.

### 2.3.3.2 Identification of Test Conditions

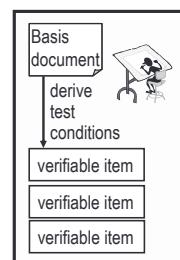
The features to be tested mentioned in the test design can be expressed as test conditions or test requirements. A test condition is a verifiable item or element.



The nature of a test condition depends on the nature of the test basis documentation. It may for example be a function, a transaction, a feature, a requirement, or a structural element like an interface parameter or a statement in the code.

The test conditions are based on or identical to our coverage items. They are the items we are covering when we test the test object.

We cannot expect to be able to cover 100% of all the relevant coverage items for our test. This is where we take the completion criteria into account in our specification work.



The completion criteria often include the percentage of the coverage items we must cover, called the coverage. We select the test conditions to get the highest coverage. Prioritization criteria identified in the risk analysis and test planning may be applied in the analysis activity to pick out the most important coverage items if we cannot cover them all.

The completion criteria for a component test could include a demand for 85% decision coverage.



If we are lucky the test conditions are clearly specified and identifiable in the test basis documentation, but in many cases it can be quite difficult. The earlier testers have been involved in the project, the easier this task usually is.

The documentation of a test condition must at least include:

- ▶ Unique identification
- ▶ Description
- ▶ Reference to test basis documentation, if not taken from there directly

**Ex.**

The example here is based on the EuroBonus scheme of StarAlliance. This short description is taken from the SAS Web site:

*There are 3 member levels: Basis, Silver, Gold.*

*Your member level is determined by the number of Basis Points you earn within your personal 12-months period. You will automatically be upgraded to Silver Member if you earn 20.000 Basis Points during your earning period.*

*If you earn 50.000 Basis Points in the period, you become a Gold Member. The earning period runs from the first day in the joining month and 12 months forward.*

Some of the test conditions that which can be extracted from this are:

- 1) When the sum of basis points is less than 20.000, the member status is Basis.
- 2) When the sum of basis points is equal to or greater than 20.000, the member level is set to Silver.
- 3) When the sum of basis points is equal to or greater than 50.000, the member level is set to Gold.

There are many more—and just as many questions to be posed!



Only if the test conditions are not clearly defined in the basis documentation do we have to document them ourselves. If we do so we must *get the test conditions reviewed* and approved by the stakeholders.

### 2.3.3.3 Creation of Test Cases

Based on the test conditions, we can now produce our first high-level test cases and subsequently low-level test cases.



A high-level test case is a test case without specific values for input data and expected results, but with logical operators or other means of defining what to test in general terms.

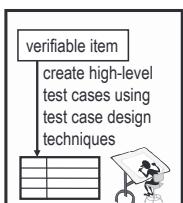
The test cases we design should strike the best possible balance between being:

- ▶ Effective: Have a reasonable probability of detecting errors
- ▶ Exemplary: Be practical and have a low redundancy
- ▶ Economic: Have a reasonable development cost and return on investment
- ▶ Evolvable: Be flexible, structured, and maintainable

The test case design techniques make it possible to create test cases that satisfy these demands.

The test techniques help us identify the input values for the test cases.

*The techniques cannot supply the expected result.*



We use appropriate test case design technique(s) as specified in the test level plan to create the high-level test cases. Test case design techniques are discussed in Chapter 4.



The documentation of a *test case* at this stage must at least include:

- ▶ Unique identification
- ▶ Description
- ▶ References to test condition(s) on which the test case is based and to test design(s) to which the test case belongs

There may well be many-to-many relationships between test conditions and high-level test cases and/or between high-level test cases and test designs.

Even though IEEE is quite specific in its requirements for the test specification it is not very often that test conditions and high-level test cases are officially documented. They are usually sketched out during the analysis and design work. Only the test designs and their procedures and low-level test cases are kept in the test specification. The decision about how much documentation of test conditions and high-level test cases to keep must be based on the strategy and the risks involved.



From the test conditions in the earlier example we can design the following high-level test cases using the equivalence partitioning technique:



- HTC 1) Check that a negative sum of basis points is not allowed.
- HTC 2) Check that a sum of basis points of less than 20.000 will give a membership level basis.
- HTC 3) Check that a sum of basis points of more than 20.000 and less than 50.000 will give a membership level silver.
- HTC 4) Check that a sum of Basis Points of more than 50.000 will give a membership level gold.

The analysis of the basis documentation will also reveal requirements concerning the test environment, not least the required test data. The test environment should be specified to a sufficient level of details for it to be set up correctly; and it should be specified as early as possible for it to be ready when we need it. Test environment requirements are discussed later.



From the high-level test cases we go on to define the low-level test cases. It is not always possible to execute all the test cases we have identified; the actual test cases to be executed must be selected based on the risk analysis.



A low-level test case is a test case with specific values defined for both input and expected result.

The documentation of a *low-level test case* must at least include:



- ▶ Unique identification
- ▶ Execution preconditions
- ▶ Inputs: data and actions
- ▶ Expected results including postconditions
- ▶ Reference(s) to test conditions and/or directly to basis documentation

**Ex.**

One low-level test case created from the list of these high-level test cases could be:

ID	Precondition	Input	Expected result	Postcondition
15.2	The current sum of basis points for Mrs. Hass is 14.300 The system is ready for entry of newly earned basis points for Mrs. Hass.	Enter 6.500 Press [OK]	The sum is shown as 20.800 The member status is shown as silver	The system is ready for a new member to be chosen.



The expected result must be determined from the basis documentation where the expectations for the coverage items are described. *The expected result must never, ever be derived from the code!*

The expected results should be provided in full, including not only visible outputs but also the final state of the software under testing and its environment. This may cover such factors as changed user interface, changed stored data, and printed reports.

We may, for example, have the following test cases, where the first gives a visible output and the second does not give a visible output, but makes a new form current.

**Ex.**

Case	Input	Expected result
1.	Enter "2" in the field "Number of journeys:"	Value in the field "Total points:" is the value in field "Points per journey:" x 2.
2.	Try to enter "10" in the field "Number of journeys:"	Value in the field "Total points:" is unchanged. The error message pop-up is current and showing error message no. 314.

In some situations the expectations may not be formally specified. Therefore it is sometimes necessary to identify alternative sources, such as technical and/or business knowledge. RAD is a particular example of where the requirements may not be formally specified.

If it turns out that it is not possible to identify what to test against, you must *never, ever just guess or assume*. Nothing to test against entails no test!

Sometimes it can be difficult to determine the expected result from the basis documentation. In such cases an oracle may be used. Oracles are discussed under tools in Section 9.3.2.

It cannot be pointed out strongly enough that if you guess about what to test and go ahead with the test specification based on your assumptions and guesses, *you are wasting everybody's time*. The chance of your getting it right is not high.

You also prevent your organization from getting better, because the people responsible for the source documentation will never know that they could easily do a better job. Go and talk to the people responsible for the source documentation. Point out what you need to be able to test. Make suggestions based on your test experience. Use some of the methods from test techniques to express the expectations, for example decision tables. Help make the source documentation better.



### 2.3.4 Requirements

This book is about testing, not requirements. A short introduction to requirements is, however, given in this section. The purpose of this is to make testers understand requirements better, and equip them to take part in the work with the requirements and to express test-related requirements for the requirements produced for a product.

All product development starts with the requirements. The higher level testing is done directly against requirements. The lower level testing is done against design that is based on the requirements. All testing is hence based on the requirements.



#### 2.3.4.1 Requirement Levels

Requirements should exist at different levels, for example:

- ▶ Business requirements
- ▶ User requirements
- ▶ System requirements



Requirements are rooted in or belong to different stakeholders. Different stakeholders speak different "languages" and the requirements must be expressed to allow the appropriate stakeholders to understand, approve, and use them.

The organization and top management “speak” money—they express business requirements. Business requirements may be tested, but most often they are not tested explicitly.

The users speak “support of my work procedures”—they express user requirements. User requirements are tested in the acceptance testing.

Following a possible product design, where the product is split up in, for example, a software system and a hardware system, we must express the system requirements. The software requirements are for the software developers and testers, and they are tested in the system testing.

### 2.3.4.2 Requirement Types

The requirement specification at each level must cover all types of requirements.

The most obvious requirements type is functional. No functionality entails no system. But as important as it may be, the functionality is not enough.

We must have some requirements expressing how the functionality should behave and present itself. These requirements are usually known as nonfunctional requirements. We could also call them functionality-supporting requirements. These requirements are discussed in detail in Chapter 5.

The functional and nonfunctional requirements together form the product quality requirements.

On top of this we may have environment requirements. These are requirements that are given and cannot be discussed. They can come both from inside and outside of the organization and can be derived from standards or other given circumstances. Environment requirements may, for example, define the browser(s)

that a Web system must be able to work on, or a specific standard to be complied with.

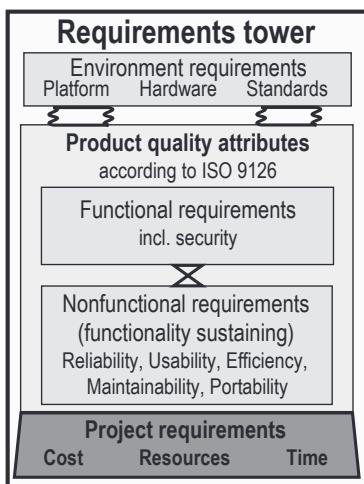
To make the requirements tower balance we need to have project requirements (or constraints) to carry the other requirements. These are cost-, resources-, and time-related, and the worry of the project management.

### 2.3.4.3 Requirement Styles

Requirements can be expressed in many ways. Typical styles are:



- ▶ Statements
- ▶ Tasks
- ▶ Models
- ▶ Tables



The most common style is the statement style. Here each requirement is expressed as a single (or very few) sentences in natural language. Some rules or recommendations should be observed when expressing requirements in statements:

- ▶ Start with: "The product shall ..." —to keep focus on the product or system
- ▶ Avoid synonyms—stick to a defined vocabulary
- ▶ Avoid subjective words (useful, high, easy)—requirements must be testable!
- ▶ Avoid generalities like "etc." "and so on"—this is impolite; think the issue through
- ▶ Be aware of "and" and "or"—is this really two or more requirements?

To make statement requirements more precise and testable we can use metrics and include information such as the scale to use, the way to measure, the target, and maybe acceptable limits. This is especially important for non-functional requirements!

Examples of such requirements (with unique numbers) are:

[56] The maximum response time for showing the results of the calculation described in requirements 65 shall be 5 milliseconds in 95% of at least 50 measurements made with 10 simultaneous users on the system.

[UR.73] It shall take a representative user (a registered nurse) no more than 30 minutes to perform the task described in use case 134 the first time.

Ex.

A *task* is a series of actions to achieve a goal. Task styles may be stories, scenarios, task lists, or use cases. Requirements expressed in these ways are easy to understand, and they are typically used to express user requirements. They are easy to derive high-level test cases and procedures from.

A *model* is a small representation of an existing or planned object. Model styles may be domain models, prototypes, data models, or state machines.

A *table* is a compact collection and arrangement of related information. Tables may be used for parameter values, decision rules, or details for models.

The styles should be mixed within each of the requirement specifications so that the most appropriate style is always chosen for a requirement.

The collection of requirements for each level documented in the requirement specification is in fact a model of the product or the system. This model is the one the test is based on.

### 2.3.5 Traceability

References are an important part of the information to be documented in the test specification. A few words are needed about these.

There are two sets of references:

- ▶ References between test specification elements
- ▶ References from test specification elements to basis documentation

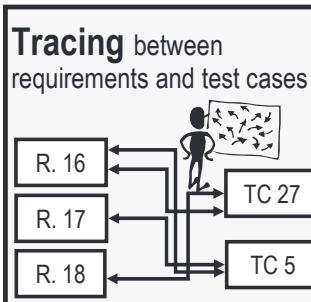
The first set of references describes the structure of the elements in the test specification. These may be quite complex with, for example, test cases belonging to more test procedures and more test groups.

The references to the basis documentation enable traceability between what we are testing and how we are testing it. This is very important information. Ultimately traceability should be possible between test cases and coverage items in the basis documentation.



*Traces should be two-way.*

You should be able to see the traces from the test cases to the covered coverage items. This will help you to identify if there are test cases that do not trace to any coverage item—in which case the test case is superfluous and should be removed (or maybe a specification like a requirement or two should be added!). This “backward” trace is also very helpful if you need to identify which coverage item(s) a test case is covering, for example, if the execution of the test case provokes a failure.



You should also be able to see the traces from the coverage items to the test cases. This can be used to show if a coverage item has no trace, and hence is not covered by a test case (yet!). This “forward” trace will also make it possible to quickly identify the test case(s) that may be affected if a coverage item, say a requirement, is changed.

If the coverage items and the test cases are uniquely identified, preferably by a number, it is easy to register and use the trace information.

Instead of writing the trace(s) to the coverage item(s) for each test case, it is a good idea to collect the trace information in trace tables. This can be done in the typical office automation system, such as in a Word table, Excel, or (best) a database.

The example on the opposite page is an extract of two tables, showing the “forward” and the “backward” traces between test cases and requirements, respectively.

Requirements to Test Cases	From Test Cases to Requirements	Ex.
<b>9.1.1.5 (8)</b>	<b>3.1 (7)</b>	
2.5 (45)	10.2.1.3.a (74)	
<b>9.1.2.1.a (14)</b>	10.6.2.7.a (123)	
5.1 (10)	10.6.2.7.b (124)	
<b>9.1.2.1.b (15)</b>	10.6.2.10.c (131)	
5.3 (13)	<b>3.2 (36)</b>	
5.4 (14)	10.5.1.1 (98)	
5.5 (12)	10.5.1.3 (100)	

### 2.3.6 Metrics for Analysis and Design

Metrics to be defined for the monitoring and control of the test analysis and design activities may include:

- ▶ Number of specified test conditions and high-level requirements over time
- ▶ Coverage achieved in the specification (for example, for code structures, requirements, risks, business processes), over time
- ▶ Number of defects found during analysis and design
- ▶ Other tasks commenced and completed over time, for example, in connection with test environment specifications
- ▶ Time spent on each task over time

This will, of course, have to be compared to the estimates and schedule of the test analysis and design activities.

## 2.4 Test Implementation and Execution

The purpose of the test implementation is to organize the test cases in procedures and/or scripts and to perform the physical test in the correct environment.



The inputs on which this process is based are:

- ▶ Level test plan
- ▶ Test conditions and test design
- ▶ Other relevant documents
- ▶ The test object



The activities are:

- ▶ Organizing test procedures
- ▶ Design and verify the test environment
- ▶ Execute the tests



This is the first place from which iterations may occur

- ▶ Record the testing
  - ▶ Check the test results
-  The output consists of:
- ▶ Test specification
  - ▶ Test environment
  - ▶ Test logs
  - ▶ Incident reports
  - ▶ Tested test object

### 2.4.1 Input to Test Implementation and Execution

The input from the level test plan that we need for this process is:

- ▶ Scheduling and staffing for the activities
- ▶ Definition of the test object(s)
- ▶ Specification of test environment
- ▶ Entry criteria for the test execution
- ▶ Exit criteria, including coverage criteria

From the test analysis and design process we need the test specification in its current state.

We might need other documentation, for example, a user manual, documentation of completion of preceding test work, and logging sheets. For the actual execution of the test we obviously need the test object.

### 2.4.2 Documentation of Test Implementation and Execution

The test specification is finished in this process where the test procedures are laid out. During this work the requirements concerning the test environment are finalized.

The test environment must be established before the test execution may start. In some cases the test environment is explicitly documented.

The test execution is documented in test logs. When failures occur these should be documented in incident reports.

### 2.4.3 Activities in Test Implementation and Execution

#### 2.4.3.1 Organizing Test Procedures

The low-level test cases should now be organized and assembled in test procedures and/or test scripts.

The term “procedure” is mostly used when they are prepared for manual test execution, while the term “script” is mostly used for automatically executable procedures.



The degree of detail in the procedures depends on who will be executing the test. They should therefore always be written with the intended audience in mind. Experienced testers and/or people with domain knowledge and knowledge about how the system works will need far less details than “ignorant” testers.

What we need to specify here is the actual sequence in which the test cases should be executed.

The documentation of a *test procedure* must at least include:

- ▶ Unique identification
- ▶ Description
- ▶ References to high-level test cases and/or to test conditions and/or directly to basis documentation to be covered by the procedure
- ▶ An explicit description of the preconditions to be fulfilled before the actual test execution can start
- ▶ Included low-level test cases

Test procedures may be organized in such a way that the execution of one test procedure sets up the prerequisites for the following. It must, however, also be possible to execute a test procedure in isolation for the purpose of confirmation testing and regression testing. The prerequisites for a test procedure must therefore always be described explicitly.

Test procedures may be hierarchical, that is “call others,” for example, generic test cases.

The test groups and the specification of their test procedures must be revisited to ensure that they are organized to give a natural flow in the test execution. Remember that the production of the test specification is an iterative process. We need to keep on designing and organizing test cases, test procedures, and test groups until everything falls into place and we think we have achieved the required coverage.

The organization in test procedures could be looked at as the execution schedule. It could be fixed, but it could also be dynamic. For specific purposes, especially for regression testing, some of the test procedures may be selected and reorganized in other execution schedules that fit the specific purpose.

A test procedure should not include too many or too few test cases—a maximum of 20 test cases and a minimum of 2–4 test cases is a good rule of thumb.

The test procedure may also include facilities for logging the actual execution of the procedure.

There are many ways to lay out the specification of test procedures and test cases. It is a good idea to set up a template in the organization.



**Ex.**

Here is an example of a template for a test procedure. The procedure heading contains fields for the required information and fields to allow the procedure to be used for logging during test execution. The template for the cases contains unique numbering of the case (within the procedure), input and expected result, and a column for registration of the actual result to be used for logging during execution.

Test procedure: n.n (n)

<b>Test procedure:</b>			
<b>Purpose:</b> This test procedure tests ...			
Traces:			
<b>Prerequisites:</b> Set up ...			
<b>Expected duration:</b> x minutes			
<b>Execution information</b>			
<b>Test date and time:</b>		<b>Initials:</b>	
<b>Test object identification:</b>		<b>Result:</b>	
Case	Input	Expected result	Actual result
1.			
2.			

Note that the template indicates a unique identification of the test procedure (n), and a number indicating its position among all the other test procedures (n.n).

To facilitate estimation the test designer is required to provide an estimate of the execution time for manual execution of the test procedure.

### ***Quality Assurance of the Test Specification***

Before the test specification is used in the test execution it should be reviewed. The review should ensure that the test specification is correct with respect to the test basis, including any standards, that it is complete with respect to the required coverage, and that it can be used by those who are going to execute the test.

Apart from the obvious benefits of having the test specification reviewed, it also has some psychological benefits. Usually we as testers review and test the work products of the analysts and developers, and we deliver feedback in



the form of verbal or written review reporting and incident reports.

This may make us seem as those who are always the bearers of bad news and ones who never make any mistakes ourselves. Getting the analysts and developers to review our work will reverse those roles; it will make us learn what it is like to receive feedback, and it will make the analysts and developers learn what it is like to deliver feedback and learn that even testers make mistakes!

The review may be guided by a checklist, of which a very small example is shown here:

- ▶ Is the test specification clear and easily understood?
- ▶ Is the test structure compatible with automated test?
- ▶ Is it easy to maintain?
- ▶ Is it easy for others to perform a technical review?



#### 2.4.3.2 Test Environment Specification and Testing

The test environment is a necessary prerequisite for the test execution—without a proper environment the test is either not executable at all or the results will be open to doubt.

The environment is first outlined in the test plan based on the strategy. The test plan also describes by whom and when the test environment is to be created and maintained. Some additional requirements for the environment may be specified in the test specification in the form of prerequisites for the test procedures, and especially for test data. The exact requirements for test data needed to execute test procedures may only be determined quite close to the actual execution. It is very important that planning and facilities for setting up specific test data are made well in advance of the execution.

The description of the test environment must be as specific as possible in order to get the right test environment established at the right time (and at the right cost). Beware: *The setting up of the test environment is often a bottleneck* in the test execution process, mostly because it is insufficiently described, underestimated, and/or not taken seriously enough. Either the environment is not established in time for the actual test execution to begin and/or it is not established according to the specifications. If the test environment is not ready when the test object is ready for the test to be executed, *it jeopardizes the test schedule*. If it is not correct, *it jeopardizes the trustworthiness of the test*.



The descriptions of the test environment must cover:

- ▶ Hardware—to run on and/or to interface with
- ▶ Software—on the test platform and other applications
- ▶ Peripherals (printers including correct paper, fax, CD reader/burner)



- ▶ Network—provider agreements, access, hardware, and software
- ▶ Tools and utilities
- ▶ Data—actual test data, anonymization, security, and rollback facilities
- ▶ Other aspects—security, load patterns, timing, and availability
- ▶ Physical environment (room, furniture, conditions)
- ▶ Communication (phones, Internet, paper forms, paper, word processor)
- ▶ Sundry (paper, pencils, coffee, candy, fruit, water)



*Problems with the test environment may force testing to be executed in other less suitable environments.* The testing could be executed in inappropriate competition with other teams and projects. If we test in the development environment, test results can be unpredictable for inexplicable reasons due to the instability of this environment. In the worst case, testing is executed in the production environment where the risk to the business can be significant.

The specific requirements for the test environment differ from test level to test level. The test environment must, at least for the higher levels of testing, be as realistic as possible, that is it should reflect the future production environment.

The need for the environment to reflect the production environment is not as pronounced for the lower test levels. In component testing and integration testing the specification must, however, include requirements concerning any drivers and stubs.



It may in some cases be *too expensive, dangerous, or time-consuming* to establish such a test environment. If this is the case the test may be un-executable and other test methods, like inspection of the code, may be used to verify the product.



As the testers we are, we have to verify that the test environment is complete according to the specifications and that it works correctly before we start to execute our test procedures. We must ensure that the test results we get are valid, that is if a test passes, it is because the test object is correct, and if it fails it is because the test object, and not the test environment, has a defect—and vice versa.



#### 2.4.3.3 Checking Execution Entry Criteria

Even though we are eager to start the test execution we should not be tempted to make a false start. We need to make sure that the execution entry criteria are fulfilled.



If the test object has not passed the entry criteria defined for it, do not start the test execution. You will waste your time, and you risk teaching the developers or your fellow testers that they don't need to take the entry criteria seriously.

We of course also need to have the people taking part in the test execution available, as specified in the test plan. The test executors must be appropriately trained, and any stakeholders needed, for example, customers to witness the execution, must be present and briefed.

Efficient and timely execution of the tests is dependent on the support processes being in place. It is particularly important that the configuration management is working well, because of the interfaces between the testing process and the configuration management process, including:

- ▶ The ability to get the correct version of the test object, the test specification, and/or the ability to get the correct versions of any other necessary material
- ▶ The ability to be able to report the failures and other incidents found during the testing
- ▶ The ability to follow the progress of the failures and plan any necessary confirmation testing and regression testing
- ▶ The ability to register approval of successful removal of failures



Support processes are discussed in Chapter 1.



#### 2.4.3.4 Test Execution

The execution of the tests is what everybody has been waiting for: the moment of truth!

In structured testing, as we have discussed earlier, in principle all the testers have to do during test execution is to follow the test specification and register all incidents on the way. If the execution is done by a tool, this is exactly what will happen.

We have taken great care in writing the test procedures, and it is important to follow them. There are several reasons for this:



- ▶ We need to be able to trust that the specified testing has actually been executed.
- ▶ We need to be able to collect actual time spent and compare it with the estimates to improve our estimation techniques.
- ▶ We need to be able to compare the progress with the plan.
- ▶ We need to be able to repeat the tests exactly as they were executed before for the sake of confirmation testing and regression testing.
- ▶ It should be possible to make a complete audit of the test.

None of this is possible if we don't follow the specification, but omit or add activities as we please.

There is nothing wrong with getting new ideas for additional test cases

to improve the test specification during the execution. In fact we neither can, nor should, avoid it. But new ideas must go through the right channels, not just be acted out on the fly. The right channel in this context is an incident management system. New ideas for tests should be treated as incidents (enhancement requests) for the test. This is another reason why it is important to have the configuration management system in place before the test execution starts.

 It is quite possible that some of the test execution time has been reserved for performing experienced based testing, where we don't use prespecified test procedures. These techniques are discussed in Section 4.4.

#### 2.4.3.5 Identifying Failures

For each test case we execute the actual result should be logged and compared to the expected result, defined as part of the test case. This can be done in various ways depending on the formality of the test. For fairly informal testing a tick mark,  $\checkmark$ , is sufficient to indicate when the actual result matched the expected result. For more formal testing, for example, for safety-critical software, the authorities require that the actual result is recorded explicitly. This could be in the form of screen dumps, included reports, or simply writing the actual result in the log. This type of logging may also serve as part of the proof that the test has actually been executed.

 We need to be very careful when we compare the expected result with the actual result, in order not to miss failures (called false positives) or report correct behavior as failures (called false negatives).

 If the actual outcome does not comply with the expected outcome we have a failure on our hands. Any failure must be reported in the incident management system. The reported incident will then follow the defined incident life cycle. Incident reporting and handling is discussed in Chapter 7.

 It is worth spending sufficient time reporting the incident we get. Too little time spent on reporting an incident may result in wasted time during the analysis of the incident. In the worst case it may be impossible to reproduce the failure, if we are not specific enough in reporting the circumstances and the symptoms.

 Don't forget that the failure may be a symptom of a defect in our work products, like the test environment, the test data, the prerequisites, the expected result, and/or the way the execution was carried out. Such failures should also be reported in order to gather information for process improvement.

#### 2.4.3.6 Test Execution Logging

As we execute, manually or by the use of tools, we must log what is going on. We must record the precise identification of what we are testing and the test

environment and test procedures we use. We must also log the result of the checking, as discussed above. Last but not least we must log any significant event that has an effect on the testing.

The recording of this information serves a number of purposes. It is indispensable in a professional and well-performed test.

The test execution may be logged in many different ways, often supported by a test management tool. Sometimes the event registration is kept apart in a test journal or diary.

The IEEE 829 standard suggests the following contents of a test log:

Test log identifier

1. Description of the test
2. Activity and event entries

It is handy and efficient if the test specification has built-in logging facilities that allow us to use it for test recording as we follow it for test execution. An example of this is shown here.



<b>Test Procedure: 3.6 (17)</b>			
<b>Purpose:</b> This test suite tests ...			
<b>Rationale:</b> User requirement 82			
<b>Prerequisites:</b> The form ...			
<b>Expected duration:</b> 15 min.			
<b>Execution time:</b> Log when <b>Initials:</b> Log who			
<b>System:</b> Identify object etc. <b>Result:</b> Log overall result			
Case	Input	Expected output	Actual output
1.	Enter...		<b>Log result</b>

The information about which test procedures have been executed and with what overall result must be available at any given time. This information is used to monitor the progress of the testing.

The identification of the test object and the test specification may be used to ensure that possible confirmation testing after defect correction is done on the correct version of the test object (the new version) using the correct version of the test specification (the old or a new as the case might be).

The rationale—the tracing to the coverage items—can be used to calculate test coverage measures. These are used in the subsequent checking for test completion.

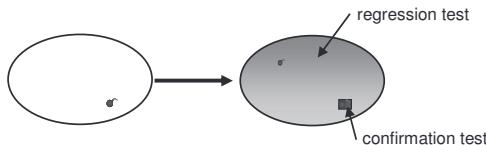
Information about who executed the test may be useful in connection with defect finding, for example, if it turns out to be difficult for the developer to reproduce or understand the reported failure.

#### 2.4.3.7 Confirmation Testing and Regression Testing

During testing we get failures. In most cases the underlying defects are corrected and the corrected test object is handed over to the testers for confirmation. This is the situation where we iterate in the test process and go back to the test execution process. We go back to perform confirmation testing and regression testing.

Confirmation testing and regression testing are important activities in test execution. They can appear in all the test levels from component testing to (one hopes rarely) acceptance testing and even during maintenance of a product in operation.

These two types of change-related testing have one thing in common: they are executed after defect correction. Apart from that, they have very different goals.



In the figure above the test object with a defect is shown to the left. The defect has been unveiled in the testing. The defect has subsequently been corrected and we have got the new test object back again for testing; this is the one to the right.

What we must do now are confirmation testing and regression testing of the corrected test object.

##### *Confirmation Testing*

Confirmation testing is the first to be performed after defect correction. It is done to ensure that the defect has indeed been successfully removed. The test that originally unveiled the defect by causing a failure is executed again and this time it should pass without problems. This is illustrated by the dark rectangle in the place where the defect was.

##### *Regression Testing*

Regression testing may—and should—then be performed.

Regression testing is repetition of tests that have already been performed without problems to ensure that defects have not been introduced or uncovered as a result of the change. In other words it is to ensure the object under test has not regressed.

**Ex.**

This example shows a case of regression: A correction of a fault in a document using the “replace all” of the word “Author” with the word “Speaker” had an unintended effect in one of the paragraphs:

“... If you are providing the Presentation as part of your duties with your company or another company, please let me know and have a Speakerized representative of the company also sign this Agreement.”

The amount of regression testing can vary from a complete rerun of all the test procedures that have already passed, to, well, in reality, no regression testing at all. The amount depends on issues such as:

- ▶ The risk involved
- ▶ The architecture of the system or product
- ▶ The nature of the defect that has been corrected

The amount of regression testing we choose to do must be justified in accordance with the strategy for the test.

Regression testing should be performed whenever something in or around the object under testing has changed. Fault correction is an obvious reason. There could also be others, more external or environmental changes, which could cause us to consider regression testing.

An example of an environment change could be the installation of a new version of the underlying database administration system or operating system. Experience shows that such updates may have the strangest effects on systems or products previously running without problems.

Ex.

#### 2.4.4 Metrics for Implementation and Execution

Metrics to be defined for the implementation and execution of the test implementation and execution activities may include:

- ▶ Number of created test environments over time
- ▶ Number of created test data over time
- ▶ Number of created test procedures over time
- ▶ Number of initiated test procedures over time
- ▶ Number of passed test procedures over time
- ▶ Number of failed test procedures over time
- ▶ Number of passed confirmation tests over time
- ▶ Number of test procedures run for regression testing over time
- ▶ Time spent on the various tasks

This will, of course, have to be compared to the estimates and schedule of the test implementation and execution activities.

### 2.5 Evaluating Exit Criteria and Reporting

Test execution, recording, control, retesting, and regression testing must be

continued until we believe that the exit criteria have been achieved. All the way we need to follow what is going on.



The purpose of the test progress and completion reporting is to stay in control of the testing and deliver the results of the testing activities in such ways that they are understandable and useful for the stakeholders.



The inputs on which this process is based are:

- ▶ Test plan
- ▶ Measurements from the test development and execution processes



The activities are:

- ▶ Comparing actual measurements with estimates and planned values
- ▶ Reporting test results



The output consists of:

- ▶ Presentation of test progress
- ▶ Test report



This is the second place from which iterations may occur

### **2.5.1 Input to Test Progress and Completion Reporting**

The input from the level test plan that we need for this process is:

- ▶ Scheduling and staffing for the activities
- ▶ Exit criteria



### **2.5.2 Documentation of Test Progress and Completion Reporting**

The documentation of the progress must be presented in various ways according to who is receiving it. The audience may be the customer, higher management, project management and participants, and testers.

Section 3.4.2 discusses presentation of monitoring information in great detail.



At the completion of each test level a test summary report should be produced. The ultimate documentation of completion is the final test summary report for the entire test assignment. The contents of a test summary report are described in Section 3.2.3.5.

### **2.5.3 Activities in Test Progress and Completion Reporting**

The activities related to the test progress and completion reporting are discussed in the sections referenced above.

### 2.5.3.1 Checking for Completion

A check against the test exit criteria is mandatory before we can say that the testing is completed at any level. To warrant a stop it is important to ensure that the product has the required quality.

The exit criteria are tightly connected to the coverage items for the test, the test case design techniques used, and the risk of the product. The exit criteria therefore vary from test level to test level.

Examples of exit criteria are:

- ▶ Specified coverage has been achieved
- ▶ Specified number of failures found per test effort has been achieved
- ▶ No known serious faults
- ▶ The benefits of the system as it is are bigger than known problems



If the exit criteria are not met the test cannot just be stopped. An iteration in the test process must take place: We have to go back to where something can be repeated to ensure that the exit criteria are fulfilled.

In most cases additional test procedures are required. This means that the test analysis and design process must be revisited and more test cases and procedures added to increase coverage. These test procedures must then be executed, and the results recorded and checked. Finally the checking of the exit criteria must be completed.

Alternatively, the test plan may be revised to permit the relaxation (or strengthening) of test exit criteria.

*Any changes to the test completion criteria must be documented*, ideally having first identified the associated risk and agreed to the changes with the customer. Changing the test plan by adjusting the completion criteria should be regarded as an emergency situation and be very well accounted for.



When all test completion criteria are met and the report approved, the test object can be released. Release has different meanings at different points in the development life cycle:

- ▶ When the test is a static test the test object (usually a document) can be released to be used as the basis for further work.
- ▶ When the test is a test level for dynamic test the test object is progressively released from one test level to the next.
- ▶ Ultimately the product can be released to the customer.

### 2.5.4 Metrics for Progress and Completion Reporting

Metrics to be defined for the progress and control activities themselves may include:

- ▶ Number of tasks commenced over time
- ▶ Task completion percentage over time
- ▶ Number of task completed over time
- ▶ Time spent on each task over time

This will of course have to be compared to the estimates and schedule of the test progress and completion activities.

## 2.6 Test Closure



The purpose of the test closure activities is to consolidate experience and place test ware under proper control for future use.



- The inputs on which this process is based are:
- ▶ Level test plan
  - ▶ Test ware, including test environment



- The overall procedure consists of the activities:
- ▶ Final check of deliveries and incident reports
  - ▶ Secure storage/handover of test ware
  - ▶ Retrospection



- The output generated in this process is:
- ▶ Test experience report
  - ▶ Configuration management documentation

### 2.6.1 Input to Test Closure

The input from the test plan that we need for this process is:

- ▶ Scheduling and staffing for the activities
- ▶ Planned deliveries



Furthermore we need all the test ware, both the test plans and specification, we have produced prior to test execution, the test environment, and the logs, incidents, and other reports we have produced during and after test execution. We also need the experiences made by all the participants and other stakeholders. These are often in the form of feelings and opinions of what has been going on.

### 2.6.2 Documentation of Test Closure

The documentation from this process is an experience report or a retrospective report from the retrospective meeting.

Other documentation will exist in the form it is specified in the organization's and/or customer's configuration management system.

### 2.6.3 Activities in Test Closure

#### 2.6.3.1 Check Completion Again

Before we definitively close the door to the testing assignment we need to make extra sure that we have met the part of the exit criteria. This is both in terms of test coverage and deliveries we are to produce. If this is not in order or any discrepancies not clearly documented we'll have to make sure it is before we proceed.

#### 2.6.3.2 Delivering and Archiving Test Ware

The test ware we have produced are valuable assets for the organization and should be handled carefully. For the sake of easy and economically sound future testing in connection with defect correction and development of new versions of the product we should keep the assets we have produced.

*It is a waste of time and money not to keep the test ware we have produced.*

If the organization has a well-working configuration management system this is what we must use to safeguard the test ware.

If such a system does not exist, we must arrange with those who are taking over responsibility for the product how the test ware must be secured. Those taking over could, for example, be a maintenance group or the customer.



#### 2.6.3.3 Retrospective Meeting

The last thing we have to do is to report the experiences we have gained during our testing. The measurements we have collected should be analyzed and any other experiences collected and synthesized as well. This must be done in accordance with the approach to process improvement expressed in the test policy and the test strategy, as discussed in Section 3.2.

This is also a very valuable activity since the results of the testing can be the main indicators of where processes need to be improved. This can be all processes, from development processes (typically requirements development) over support processes (typically configuration management, not least for requirements) to the test process itself.

It is important that we as testers finish our testing assignment properly by producing an experience report.



For the sake of the entire process improvement activity, and hence the entire organization, it is important that higher management is involved and asks for and actively uses the test experience reports. Otherwise, the retrospective meetings might not be held, because people quickly get engrossed in new (test) projects and forget about the previous one.



#### **2.6.4 Metrics for Test Closure Activities**

Metrics to be defined for these activities may include number of tasks commenced over time, task completion percentage over time, number of tasks completed over time, and time spent on each task over time as for the other processes.

This will of course have to be compared to the estimates and schedule of the test closure activities.

### **Questions**

1. Which three elements must always be defined for a process?
2. How do processes depend on each other?
3. What are the five activities (subprocesses) in the generic test process?
4. To which test levels and other test types does the generic test process apply?
5. Which iterations are embedded in the generic test process?
6. From where can we get inspiration for test process definitions?
7. What is the input to the test planning process?
8. What is the table of contents for a test plan suggested by IEEE 829?
9. Why is early planning a good idea?
10. What can the test basis be for each of the dynamic test levels?
11. What should be covered in the test approach description?
12. What are completion criteria?
13. What are the typical test deliveries?
14. What is the structure of a test specification according to IEEE 829?
15. What are the parameters we use to plan the test?
16. What is a work breakdown structure?
17. What are the testing roles we need to handle all test activities?
18. What are the activities in the test analysis and design process?
19. What should be in a test design according to IEEE 829?
20. What test design would be relevant for a system test?
21. What is a test condition?
22. How are test cases created?
23. What must be defined for each test case according to IEEE 829?
24. What is the expected result in a test case?
25. What could be used if the expected result cannot be determined easily?
26. What requirements types should we expect to find in a requirements specification?
27. What are the recommendations for expressing requirements as statements?
28. What is traceability?
29. What are the activities in the test implementation and execution process?
30. What is a test procedure?

31. What should be in a test procedure according to IEEE 829?
32. What are the guidelines for the length of a test procedure?
33. Why should test specifications be reviewed?
34. How can the test environment jeopardize the test?
35. What characterizes a valid test environment?
36. Why should test entry criteria be checked?
37. Which supporting process is it especially important to have in place before test execution starts, and why?
38. Why should the test specification be followed during test execution?
39. What must be done when a failure is observed?
40. What information should be recorded for each executed test procedure?
41. What are confirmation testing and regression testing?
42. How much regression testing should be done?
43. When should regression testing be performed?
44. How should test progress and completion reporting be done?
45. Why should we check for completion?
46. What can be done if the completion criteria are not met?
47. What are the activities in the test closure process?
48. Why should testware be kept?
49. What is done in a retrospective meeting?
50. What is the ultimate purpose of the experience report?



## Test Management

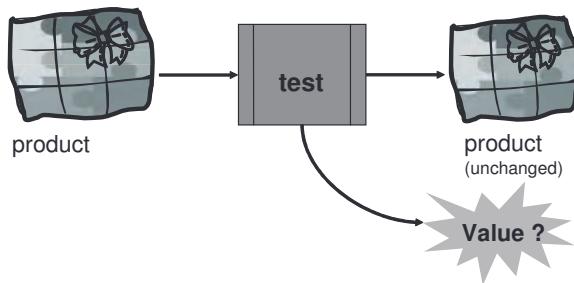
Test management is the art of planning and directing a test assignment to success. It is in many ways like project management, and yet not quite the same.

Test management must be done in close cooperation with project management, sometimes by the same person, sometimes by different people.

The test manager is the link between the test team and the development team and between the test team and higher management. It is therefore essential that the test manager is the ambassador of testing and truly understands how testing contributes to the business goals.

### 3.1 Business Value of Testing

On the face of it, testing adds no value. The product under testing is—in principle—not changed after the test has been executed.



But we are paid to test, so we must add some value to be in business. And we do!

### Contents

- 3.1 Business Value of Testing
- 3.2 Test Management Documentation
- 3.3 Test Estimation
- 3.4 Test Progress Monitoring and Control
- 3.5 Testing and Risk

*The business value of testing lies in the savings that the organization can achieve from improvements based on the information the testing provides.*

Improvements can be obtained in three places:

- ▶ The product under development
- ▶ The decisions to be made about the product
- ▶ The processes used in both testing and development

It may, however, sometimes be difficult to understand and express what the value is, both to ourselves and to others in the organization.

It is essential that test managers know and understand the value of testing and know how to express it to others to make them understand as well. Test managers must communicate the value to the testers, to other project participants, and to higher management.

Testers are often engrossed in the testing tasks at hand and don't see the big picture they are a part of; higher management is often fairly remote from the project as such and doesn't see the detailed activities.



### 3.1.1 Purpose of Testing

What testing does and therefore the immediate *purpose of testing is getting information about the product under testing*. We could say (with Paul Gerrard, founder of Aqastra): Testing is the intelligence office of the company.



The places we gather our raw data from are the test logs and the incident reports, if these are used sensibly and updated as the testing and the incident are progressing. From the raw data we can count and calculate a lot of useful quantitative information.



A few examples of such information are:

- ▶ Number of passed test cases
- ▶ Coverage of the performed test
- ▶ Number and types of failures
- ▶ Defects corrected over time
- ▶ Root causes of the failures



Most of this information is “invisible” or indigestible unless we testers make it available in appropriate formats. There is more about this in Section 3.5. This section also discusses how the information can be used to monitor the progress of the development in general and the testing in particular.

### 3.1.2 The Testing Business Case

It is not straightforward to establish a business case for testing, since we don't know in advance what savings we are going to enable. We don't know how many defects in the product we are going to unveil.

A well-established way to express the value of testing for the product is based on the cost of quality. This can be expressed as value of product improvement:

*Value of product improvement =*

(cost of failure not found – cost failure found) – cost of detection

To this we can add

*Value of decision improvement =*

(cost of wrong decision – cost of right decision) – cost of getting decision basis

*Value of process improvement =*

(cost using old process – cost using better process) – cost of process improvement

These three aspects add up to form the entire business case for testing. The aim is to get as high a value as possible.

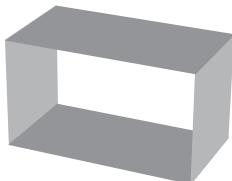
A value may be expressed either quantitatively or qualitatively. Quantitative values can be expressed in actual numbers—euros, pounds, or dollars or numbers of something, for example. Qualitative values cannot be calculated like that, but may be expressed in other terms or “felt.”

#### 3.1.2.1 The Value of Product Improvement

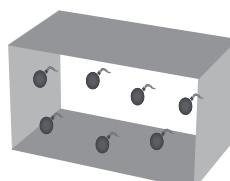
The value of product improvement is the easiest to assess.

One goal of all development is reliability in the products we deliver to the customers. Reliability is the probability that software will not cause the failure of a system for a specified time under specified conditions.

A product's reliability is measured by the probability that faults materialize in the product when it is in use.



No faults  
= 100% reliability



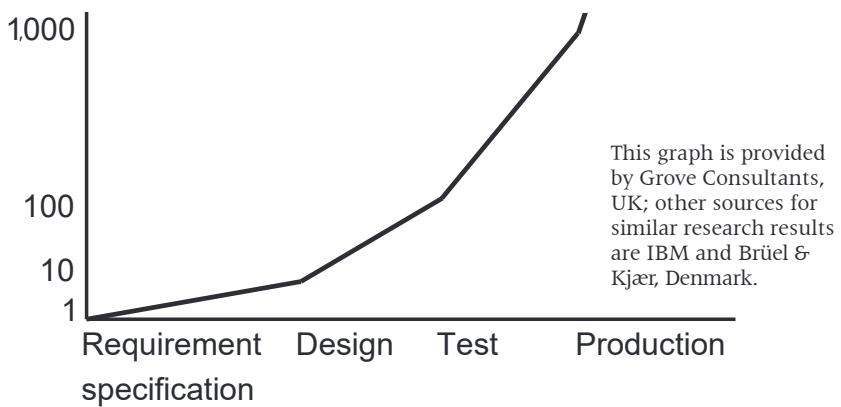
Many faults  
= x% reliability

The less failures that remain in the product we release, the higher is the reliability of the product and the lower the risk of the product failing and thereby jeopardizing its environment. Project risks range from ignorable to endangering the lives of people or companies. There is more about risk management in Section 3.6.

The earlier we get a defect removed the cheaper it is. Reviews find defects and dynamic testing finds failures, and this enables the correction of the underlying defects.

The cost of the defect correction depends on when the defect is found. Defects found and corrected early are much cheaper to correct than defects found at a later point in time. Research shows that if we set the cost of correcting a defect found in the requirements specification to 1 unit, then it will cost 10 units to make the necessary correction if the defect is first found in the design.

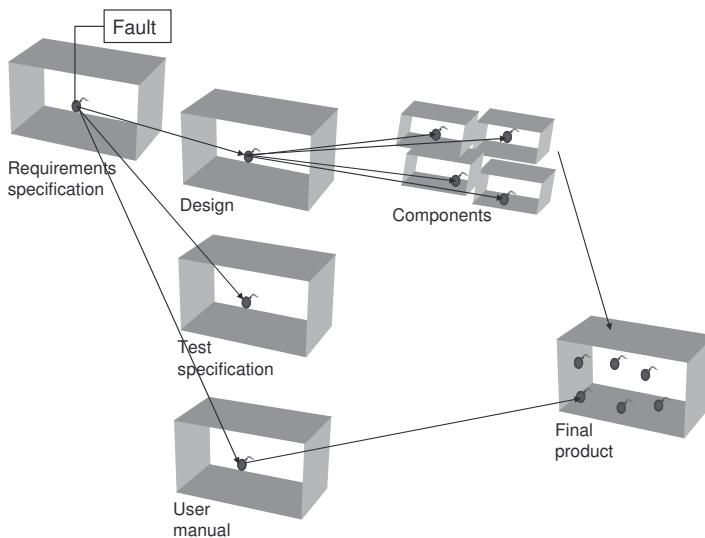
If the defect remains in the product and is not found until encountered as a failure in dynamic test, it costs 100 units to correct it. The failures found during development and testing are called internal failures, and they are relatively cheap.



If the customer gets a failure in production—an external failure—it may cost more than 1,000 units to make the necessary corrections, including the cost that the customer may incur. The analysts and programmers who can/must correct the defects may even be moved to new assignments, which are then in turn delayed because of (emergency) changes to the previous product.

The basic reason for this raise in cost is that defects in software do not go away if left unattended; they multiply. There are many steps in software

development from requirements specification to manufacturing and for each step a defect can be transformed into many defects.



There is some element of estimation in preparing the business case for product improvement. Many organizations don't know how many defects to expect, how much it costs to find defects, and how much it costs to fix them, or how much it would have cost to fix them later. The more historical data about the testing and defect correction an organization has, the easier it is to establish a realistic business case.

Let's look at a few calculation examples.

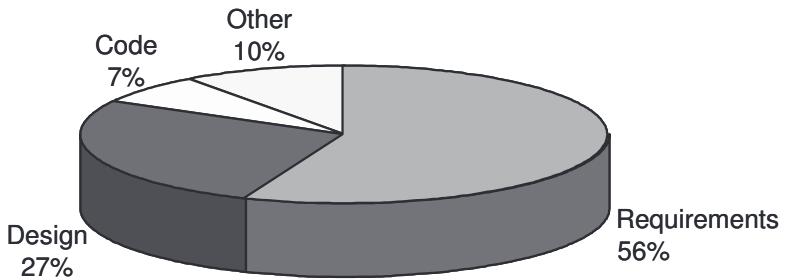
If we assume that it costs 4 units to correct a defect in the requirements phase, and 6 units to detect a defect or a failure, we can make calculations like:

Value of finding a defect in system testing rather than in production at the customer's site =  $(4,000 - 400) - 6 = 3,594$  units

Value of finding a defect in requirements specification rather than in system testing =  $(400 - 4) - 6 = 390$  units

**Ex.**

Other research show that about 50% of the defects found in the entire life of a product can be traced back to defects introduced during the requirements specification work. This is illustrated in the following figure where the origins of defects are shown.



If we combine these two pieces of research results we have a really strong case for testing, and for starting testing early on in the project!



*To get the full value of the test, it should start as early as possible in the course of a development project, preferably on day 1!*



### 3.1.2.2 The Value of Decision Improvement

From the point of view of decision making such as decisions concerning release (or not) of a product the confidence in the product and quality of the decisions are proportional to the quality and the amount of the information provided by testing. As testing progresses, more and more information is gathered and this enhances the basis for the decisions.

The more knowledge the decision makers have about what parts of the product have been tested to which depth—coverage—and which detected defects have been removed and which are still remaining, the more informed are the decisions made. The value of more informed decisions rather than less informed decisions is qualitative; it is very rarely possible to calculate this quantitatively.

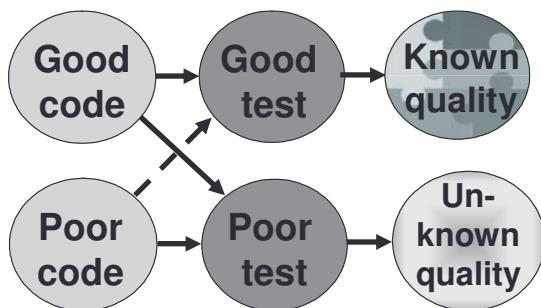


It follows from the concept of test as an information collection activity that it is not possible to test good quality into a product. But the quality of the testing reflects in the quality of the information it provides.

Good testing provides trustworthy information and poor testing leave us in ignorance.

If the starting point is a good product, a good test will provide information to give us confidence that the quality is good.

If the starting point is a poor product, a good test will reveal that the quality is low.



But if the testing is poor we will not know if we have a good or a poor product.

The line from “poor code” to “good test” is dashed, because poor coding and good testing is not often seen together. Our goal as professional test practitioners is to reduce the occurrence of poor testing.

More important decisions may also be based on the information from testing. A test report with documentation of the test and the test results can be used to prove that we fulfilled contractual obligations, if needed. It may even in some (one hopes rare) cases provide a judicial shield for the company in that it provides evidence against suits for negligence or the like. This is of qualitative value to the business.



### 3.1.2.3 The Value of Process Improvement

From the *process improvement* point of view the information gained from testing is invaluable in the analysis of how well processes fit and serve the organization. The results of such analysis can be used to identify the process that could be the subject for process improvement. The process to improve may be both the testing process and other processes.

As time goes by the information can tell us how a process improvement initiative has worked in the organization.

When the testing process improves, the number of failures sent out to the customers falls, and the organization’s reputation for delivering quality products will rise (all else being equal). The value of this is qualitative.

There is more about process improvement in Chapter 8.



## 3.2 Test Management Documentation

### 3.2.1 Overview

Proper test management requires that information about the decisions that test management makes is available and comprehensive to all stakeholders. These decisions are normally captured in a number of documents.

The test management documentation comprises:

- ▶ Test policy
- ▶ Test strategy
- ▶ Project test plan
- ▶ Level test plan



The test management documentation belongs to different organizational levels as shown in the next figure.

The *test policy* holds the organization's philosophy toward software testing.

The *test strategy* is based on the policy. It can have the scope of an organizational unit or a program (one or more similar projects). It contains the generic requirements for the test for the defined scope.

A *master test plan* is for a particular project. It makes the strategy operational and defines the test levels to be performed and the testing within those levels.

A *level test plan* is for a particular test level in a particular project. It provides the details for performing testing within a level.

The presentation of this documentation depends on the organization's needs, general standards, size, and maturity. The presentation can vary from oral (not recommended!) over loose notes to formal documents based on organizational templates. It can also vary from all the information being presented together in one document, or even as part of a bigger document, to it being split into a number of individual documents.

The more mature an organization is the more the presentation of the test management documentation is tailored to the organization's needs. The way the information is presented is not important; the information is.

### 3.2.2 Higher Management Documentation

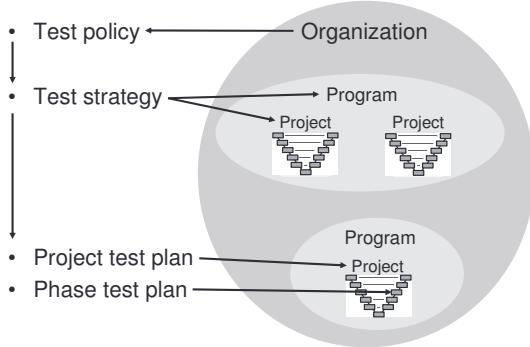
Higher management, that is management above project managers and test managers, is responsible for the two types of test management documentation discussed in this section. The documentation is used by everybody in the organization involved in testing.

#### 3.2.2.1 Test Policy

The test policy defines the organization's philosophy toward software testing. It is the basis for all the test work in the organization. A policy must be behavior-regulating in the good way—it is like a lighthouse for all the testing activities. And like every lighthouse has its own signal, every organization must have its own policy, tailored to its specific business needs.

The test policy must be short and to the point.

It is the responsibility of the top management to formulate the policy. It may, however, be difficult for top management if managers are not familiar



with professional testing, so it is often seen that the IT department (or equivalent) steps in and develops the test policy on behalf of the management.

The test policy must include:

1. Definition of testing
2. The testing process to use
3. Evaluation of testing
4. Quality targets
5. Approach to test process improvement



The test policy applies to all testing. The policy must cover all test targets. This means that there must be a policy for:

- ▶ Testing new products
- ▶ Change-related testing
- ▶ Maintenance testing

#### ***Test Policy; Definition of Testing***

The definition of testing is a brief statement formulating the overall purpose of the test in the organization.

“Checking that the software solves a business problem”  
“Activity to provide information about the quality of the products”  
“A tool box for minimization of the product risks”



#### ***Test Policy; The Testing Process***

The testing process is an overview of the activities to be performed or a reference to a full description of the testing process.

“Development and execution of a test plan in accordance with departmental procedures and user requirements found on the intranet”



Another possibility is a reference to the test process defined in standards or other literature, for example the ISTQB syllabus, the test process on which this book based. This test process was described in detail in Chapter 2.



#### ***Test Policy; Evaluation of Testing***

The evaluation of testing is the measurement to be made in order for the quality of the testing to be determined.

**Ex.**

"The number of failures reported by the field is measured every three months."

"The cost of the fault correction done after release is measured."

"The customer satisfaction is measured once a year by means of a questionnaire sent out to 200 selected customers."

### ***Test Policy; Quality Targets***

The quality targets to be achieved should be expressed so that the measurements can be used to see if we reach the goals.

**Ex.**

Examples are:

"No more than one high severity fault per 1,000 lines of delivered code to be found in the first six months of operation."

"The overall effectiveness of the test must be over 98% after the first three months in production."

"The customers must not be reporting more than three severity 1 failures during the first year of use."

"The system must not have a breakdown lasting longer than 15 minutes during the first six months in production."

### ***Test Policy; Approach to Test Process Improvement***

The organizational approach to test process improvement is the process to be used for learning from experiences. This would often be the same as the organization's general approach to software process improvement, but there might be a specific policy for the testing process improvement.

**Ex.**

"A postproject workshop where all the observations during the test process are collected shall be held within the first month after turnover to production."

"Failure reports shall be analyzed to determine any trends in the faults found in system test."

"The root cause shall be found for every severity 1 and 2 fault found during testing, and improvement actions shall be determined."

#### **3.2.2.2 Test Strategy**

The Latin word "stratagem" means a plan for deceiving an enemy in war. The enemy here is not the developers, but rather the defects! The strategy is based on the test policy and should of course be compliant with it. The strategy outlines how the risks of defects in the product will be fought. It could be said to express the generic requirements for the test. The strategy comprises the navigation rules for the testing.



The test strategy is high-level, and it should be short. It should also be readily available to all with a stake in the testing within the scope of the strategy. The strategy could be issued in a document, but it would be a good idea to present it in table form on a poster or on the intranet in the organization.

A test strategy must be for a specified scope. The scope may be all the projects in an entire organization, a specific site or department, or a program (a number of similar projects).

The overall test strategy may be chosen among the following possible approaches to the testing:

- ▶ Analytical—Using for example a risk analysis as the basis
- ▶ Model-based—Using for example statistical models for usage
- ▶ Consultative—Using technology guidance or domain experts
- ▶ Methodical—Using for example checklists or experience
- ▶ Heuristic—Using exploratory techniques
- ▶ Standard-compliant—Using given standards or processes
- ▶ Regression-averse—Using automation and reuse of scripts



There is nothing wrong with mixing the approaches. They address different aspects of testing and more approaches can support each other.

We could, for example, decide:

The component test shall be structured in compliance with the tool used for component testing.

The component integration testing shall be bottom-up integration based on a design model and in compliance with standard xxx.

The system test shall be risk-based and structured and the initial risk analysis shall be supplemented with exploratory testing.



The decisions about approaches or overall strategies have a great influence on some of the decisions that have to be made for specific topics in the strategy.

Remember that the strategy must be short—the test approach is to be refined and detailed in the test plans.

The test strategy should not be “once-written-never-changed.” As the experiences gained from finished testing activities are collected and analyzed, the results in terms of test process improvement initiatives must constantly be considered when the test strategy is formulated.



A test strategy for a defined scope could contain the following information:



#### Test strategy identifier

1. Introduction
2. Standards to use
3. Risks to be addressed
4. Levels of testing and their relationships
  - For each level, as appropriate
    - 4.1 Entry criteria
    - 4.2 Exit criteria
    - 4.3 Degree of independence
    - 4.4 Techniques to use
    - 4.5 Extent of reuse
    - 4.6 Environments
    - 4.7 Automation
    - 4.8 Measurements
    - 4.9 Confirmation and regression testing
  - 5. Incident management
  - 6. Configuration management of testware
  - 7. Test process improvement activities

#### Approvals

The numbered topics are indented as sections in the strategy. The identifier and the approvals are information about the strategy usually found on the front page.

*The strategy identifier* is the configuration management identification information for the strategy itself. It could be formed by the:



- ▶ Name of the strategy
- ▶ Organizational affiliation
- ▶ Version
- ▶ Status

This should adhere to the organization's standards for configuration management, if there is one.

#### ***Strategy; Introduction***

The introduction sets the scene for the strategy. It contains general information of use to the reader.

The introduction is usually the most organization specific chapter of the plan. It should be based on the organization's own standard. It should in any case cover:

- ▶ Purpose of the document
- ▶ Scope of the strategy
- ▶ References to other plans, standards, contracts, and so forth
- ▶ Readers' guide

### ***Strategy; Standards to Be Complied With***

In this section references to the standard(s) that the test must adhere to are provided.

Standards may be both external to the organization and proprietary standards.

*Standards are very useful.* Many people with a lot of experience have contributed to standards. Even though no standard is perfect and no standard fits any organization completely standards can facilitate the work by providing ideas and guidelines.

The more standards it is possible to reference the easier the work in the strategy, the planning, and the specification. Information given in standards must not be repeated in specific documents, just referenced.



IEEE 829, Test Documentation.  
“Test-Nice”—the company standard for test specifications.



Some appropriate standards are discussed in Chapter 8.



### ***Strategy; Risks***

The basis for the strategy can be the product risks to mitigate by the testing. Appropriate project risks may also be taken into consideration.

The strategy must include a list of the relevant risks or a reference to such a list.

Risks in relation to testing are discussed in Section 3.8.

### ***Strategy; Test Levels and Their Relationships***

The typical test strategy will include a list and description of the test levels into which the test assignments within the scope should be broken.

The levels can for example be:

- ▶ Component testing
- ▶ Component integration testing
- ▶ System testing
- ▶ System integration testing
- ▶ Acceptance testing



The levels are described in detail in Chapter 1.

The following strategy topics must be addressed for each of the levels that the strategy includes. It is a good idea to give a rationale for the decisions made for each topic, if it is not obvious to everybody.

### ***Strategy; Level Entry Criteria***

This is a description of what needs to be in place before the work in the test level can start.

The strictness of the entry criteria depends on the risk: The higher the risk the stricter the criteria.

 Ex.

An entry criterion for the system test could be that the system requirements specification has passed the first review.

### ***Strategy; Level Exit Criteria***

The testing exit or completion criteria are a specification of what needs to be achieved by the test. It is a guideline for when to stop the testing—for when it is “good enough.” It is a description of what needs to be in place before the work in the test level can be said to be finished.



Testing completion criteria represent one of the most important items in a comprehensive test strategy, since they have a great influence on the subsequent testing and the quality of a whole system.

Some completion criteria are closely linked to the chosen test case design techniques; some are linked to the progress of the test.

 ,,

The strategy does however not need to be very specific. The completion criteria will be detailed and made explicit in the test plans. A detailed discussion of completion criteria is found in Section 3.2.3.3.

The strictness of the completion criteria depends on the risk as described above.

Descriptions of the strategy for completion criteria could for example be:

 Ex.

Component test: Decisions coverage must be between 85% and 100% dependent on the criticality of the component. No known faults may be outstanding.

System test: At least 95% functional requirements coverage for priority 1 requirements must be achieved. No known priority 1 failures may be outstanding.

The test report has been approved by the project manager.

### ***Strategy; Degree of Independence***

Testing should be as objective as possible. The closer the tester is to the producer of the test object, the more difficult it is to be objective.

The concept of independence in testing has therefore been introduced.

The degree of independence increases with the “distance” between the producer and the tester. These degrees of independence in testing have been defined:

1. The producer tests his or her own product
2. Tests are designed by another nontester team member
3. Tests are designed by a tester who is a member of the development team
4. Tests are designed by independent testers in the same organization
5. Tests are designed by organizationally independent testers (consultants)
6. Tests are design by external testers (third-party testing)



As it can be seen in the list the point is who designs the test cases. In structured testing the execution must follow the specification strictly, so the degree of independence is not affected by who is executing the test. In testing with little or no scripting, like exploratory testing, the independence must be between producer and test executor.

The strategy must determine the necessary degree of independence for the test at hand. The higher the risk the higher the degree of independence.

There is more about independence in testing in Section 10.4.



### ***Strategy; Test Case Design Techniques to Be Used***

A list of the test case design techniques to be used for the test level should be provided here. The choice of test case design techniques is very much dependent on the risk—high risk: few, comprehensive techniques to choose from; low risk: looser selection criteria.

Test case design techniques could be equivalence partitioning, boundary value analysis, and branch testing for component testing.



Test case design techniques are described in great detail in Chapter 4.

### ***Strategy; Extent of Reuse***

Reuse can be a big money and time saver in an organization.

Effective reuse requires a certain degree of maturity in the organization. Configuration management needs to be working well in order to keep track of items that can be reused.

This section must provide a description of what to reuse under which circumstances.

Work product for reuse could, for example, be:

**Ex.**

- ▶ Generic test specifications
- ▶ Specific test environment(s)
- ▶ Test data

### ***Strategy; Environment in Which the Test Will Be Executed***

Generic requirements for the test environment must be given here. The specific environment must be described in the test plan, based on what the strategy states.

The requirements for the test environment depend very much on the degree of independence and on the test level at which we are working.

**Ex.**

We could for example find:

Component testing: The developer's own PC, but in a specific test area.

System test: A specific test environment established on the test company's own machine and reflecting the production environment as closely as possible.

### ***Strategy; Approach to Test Automation***

This is an area where the strategy needs to be rather precise in order for tool investments not to get out of hand.

Technical people—including testers—love tools. Tools are very useful and can ease a lot of tedious work.

*Tools also cost a lot* both in terms of money over the counter and in terms of time to implement, learn, use, and maintain. Furthermore, no single tool covers all the requirements for tool support in a test organization, and only a few tools are on speaking terms. It can be costly and risky, or indeed impossible to get information across from one tool to another.

It is important that the strategy includes a list of already existing testing tools to be used, and/or guidelines for considerations of implementation of new tools.

Test tools are described in Chapter 9.



### ***Strategy; Measures to Be Captured***

In the test policy it has been defined how the test shall be evaluated. It has also been defined what the approach to process improvement is. This governs the measures we have to collect.

Measures are also necessary to be able to monitor and control the progress of the testing. We need to know how the correspondence is between the reality and the plan. We also need to know if and when our completion criteria have been met.

Based on this, this section must contain a definition of all the metrics for testing activities. Descriptions of metrics include scales, ways of capturing the measurements, and the usage of the collected information.

Metrics and measurements are discussed in general in Section 1.3.



### ***Strategy; Approach to Confirmation Testing and Regression Testing***

Confirmation testing is done after fault correction to confirm that the fault has indeed been removed.

Regression testing should be done whenever something has changed in the product. It is done to ensure that the change has had no adverse effect on something that was previously working OK. Regression testing should follow any confirmation test; it should also for example follow an upgrade of the operation system underlying the product. The amount of regression testing to perform after a change is dependent on the risk associated with the change.

This section must outline when and how to perform confirmation testing and regression testing in the test level it is covering.



Fault correction is NOT part of the test process.

System testing: Re-execute the test case(s) that identified the fault and rerun at least 1/3 of the rest of the already executed test cases. The choice of test cases to rerun must be explained.



### ***Strategy; Approach to Incident Management***

It is hoped that a reference to the configurations management system is sufficient here.

If this is not the case it must be described how incidents are to be reported and who the incident reports should be sent to for further handling.

*Close cooperation with the general configuration management function in the organization is strongly recommended* on this. There is no need to reinvent procedures that others have already invented.



### ***Strategy; Approach to Configuration Management of Testware***

Configuration management of testware is important for the reliability of the test results. The test specification and the test environment including the data must be of the right versions corresponding to the version of product under testing. A good configuration management system will also help prevent extra work in finding or possibly remaking testware that has gone missing—something that happens all too often in testing.

Configuration management is a general support process, and if a configuration management system is in place this is of course the one the testers should use as well, and the one to which the strategy should refer.



If such a system is not in place the approach to local testing configuration management must be described. Configuration management is discussed in Section 1.1.3.



### ***Strategy; Approach to Test Process Improvement***

This could be a refinement of the approach described in the policy; see Section 3.1.1.5.

## **3.2.3 Project Level Test Management Documentation**

The two types of test management documentation discussed in this section belong to a particular project. The master test plan should be produced by the person responsible for testing on the project, ideally a test manager. The level test plans should be produced by the stakeholder(s) carrying the appropriate responsibility. This could be anybody from a developer planning a component test to the test manager planning the system or acceptance test.



The plan  
outlines the  
journey.

### **3.2.3.1 Master Test Plan**

The master test plan documents the implementation of the overall test strategy for a particular project. This is where the strategy hits reality for the first time. The master test plan must comply with the strategy; any noncompliance must be explained.

The master test plan must be closely connected to the overall project plan, especially concerning the schedule and the budget! The master test plan should be referenced from the project plan or it could be an integrated part of it.

The master test plan has many stakeholders and missions, and it must at least provide the information indicated in the following list to the main stakeholders.

<b>Stakeholder</b>	<b>Information</b>
All	Test object = scope of the test for each level Involvement in the testing activities Contribution to the testing activities Relevant testing deliverables (get/produce)
Management	Business justification and value of testing Budget and schedule
Development	Test quantity and quality Expectation concerning delivery times Entry criteria for deliverables
Test team	Test levels Schedule Test execution cycles Suspension criteria and exit criteria
Customer	Test quantity and quality

*All stakeholders in the master test plan must agree to the contents according to their interest and involvement—otherwise the plan is not valid!*

As mentioned previously, the way the information in the master test plan is presented is not important; the information is.

The detailed structure and contents of a master test plan are discussed in Section 3.2.3.3.



### 3.2.3.2 Level Test Plan

A level test plan documents a detailed approach to a specific test level, for example a component test or acceptance test. The level test plan describes the implementation of the master test plan for the specific level in even more precise detail. For instance, it would normally include a sequence of test activities, day-to-day plan of activities, and associated milestones.

The size of a level test plan depends on the level it covers—a component test plan for single components may be just 5–10 lines; system test plans may be several pages.

As for the master test plan it is vital to include all relevant stakeholders in the planning process and to get their sign-off on the plan.

### 3.2.3.3 Test Plan Template

The structure of the test plans, both the master test plan and any level test plans, should be tailored to the organization's needs.

In order not to start from scratch each time it is, however, a good idea to have a template. A template could be based on the IEEE 829 standard. This standard suggests the following contents of a test plan:



### Test plan identifier

1. Introduction (scope, risks, and objectives)
2. Test item(s) or test object(s)
3. Features (quality attributes) to be tested
4. Features (quality attributes) not to be tested
5. Approach (targets, techniques, templates)
6. Item pass/fail criteria (exit criteria including coverage criteria)
7. Suspension criteria and resumption requirements
8. Test deliverables (work products)
9. Testing tasks
10. Environmental needs
11. Responsibilities
12. Staffing and training needs
13. Schedule
14. Risks and contingencies

### Test plan approvals

The numbered topics are intended as sections in the plan; the identifier and the approvals for the plan are usually found on the front page.

*The test plan identifier* is the configuration management identification information for the test plan itself. It could be formed by the

- ▶ Name of the plan
- ▶ Organizational affiliation
- ▶ Version
- ▶ Status

This should adhere to the organization's standards for configuration management, if there is one.

### ***Test Plan; Introduction***

The introduction sets the scene for the test plan as a whole. It contains general information of use to the reader.

The introduction is usually the most organization-specific chapter of the plan. It should be based on the organization's own standard. It should in any case cover:

- ▶ Purpose of the document
- ▶ Scope of the plan, possibly including intended readership
- ▶ References to other plans, standards, contracts, and so forth
- ▶ Definitions
- ▶ Abbreviations

- ▶ Typographical conventions used
- ▶ Readers' guide

It is important to get the references precise and correct. Test planning is influenced by many aspects, including the organization's test policy, the test strategy, the development or maintenance plan, risks, constraints (time, money, resources), and the test basis and its availability and testability. References must be made to all this information—and it must be respected.

If this chapter gets too voluminous you can place some of the information in appendices.



### ***Test Plan; Test Item(s)***

Here the test object(s) or item(s) and additional information are identified as precisely and explicitly as possible. The additional information may be the appropriate source specification, for example detailed design or requirements specification, and helpful information such as the design guide, coding rules, checklists, user manual, and relevant test reports.

The test object depends on whether the plan is the master test plan or a level test plan, and in the latter case of the specific test level the plan is for.

Product XZX V2.3, based on XZX System Requirements Specification V4.2.  
The individual component: pre\_tbuly V2.3.



### ***Test Plan; Features to Be Tested***

Within the scope of the test item(s), an overview of the features to be tested is provided along with references to where the test is specified, or will be specified as the case may be. Features include both functional and nonfunctional quality attributes.

The decision about which features are to be tested and which are not is based on the applicable test strategy, the identified risks, and the mitigation activities for them. The identification of the features to be tested is also closely linked to the specified coverage items.



All functional requirements, specified in System test specification STS-XX.doc must be covered in this test.

All methods in the classes are to be tested in the component testing.



### ***Test Plan; Features Not to Be Tested***

To set the expectations of the stakeholders correctly, it is just as important to state what features are not tested as it is to state which are.

With regards to what might be expected to be tested in relation to the test item(s), we must provide a list of the features not to be tested. A reason must be given for each of the features omitted.



Performance testing is not part of this test because it will be carried out by third-party company PTESTIT. They are experts in performance testing.

In this component the function M-bladoo is not tested. It is too costly to simulate the error situation that it handles. A formal inspection has been performed on the code.

### ***Test Plan; Approach***

The test approach must be based on the strategy for the test at hand. This section expands the approach and makes it operational.

The approach must at least cover:



- ▶ The test methods and test techniques to use
- ▶ The structure of the test specification to be produced and used
- ▶ The tools to be used
- ▶ The interface with configuration management
- ▶ Measurements to collect
- ▶ Important constraints, such as availability or “fixed” deadline for the testing we are planning for.

### ***Test Plan; Item Pass/Fail Criteria***

The item pass/fail criteria are the American counterpart to what we Europeans call completion criteria. The completion criteria are what we use to determine if we can stop the testing or if we have to go on to reach the objective of the testing.



Examples of appropriate completion criteria for some test levels are:

- ▶ Component testing
  - ▶ 100% statement coverage
  - ▶ 95% decision coverage
  - ▶ No known faults
- ▶ Acceptance testing
  - ▶ 100% business procedure coverage
  - ▶ No known failures of criticality 1

### ***Test Plan; Suspension Criteria and Resumption Requirements***

Sometimes it does not make sense to persevere with the test execution. It can be a very good idea to try to identify such situations beforehand. In this section in the plan, the circumstances that may lead to a suspension of the test for a shorter or longer period are described.

More than 20% of the time is spent on reporting banal failures, caused by faults that should have been found in an earlier test phase.

 Ex.

It must also be decided and documented what must be fulfilled for the test to be resumed.

Evidence of required coverage of component testing must be provided.

 Ex.

Finally it should be stated which test activities must be repeated at resumption. Maybe every test case must be re-executed; maybe it is OK to proceed from where we stopped.

### ***Test Plan; Test Deliverables***

The deliverables are a listing and a brief description of all the documentation, logs, and reports that are going to be produced in the test process at hand. Everything must be included for the purpose of estimation and the setting of expectations.

Example of test deliverables are:

 Ex.

- ▶ Test plans
- ▶ Test specifications
- ▶ Test environment
- ▶ Logs, journals, and test reports
- ▶ Release documentation for the test object

### ***Test Plan; Testing Tasks***

This section in the plan is the work breakdown structure of the test process at hand. If we use the test process used here, it is analysis, design, implementation, execution, evaluation, reporting, and closure, all broken down into more detailed activities in an appropriate work breakdown structure. When defining the test tasks in detail it is important to remember and mention everything. Even the smallest task, which may seem insignificant, may have a significant influence on the schedule.



The tasks, together with resources and responsibilities, are input items to the test schedule.

### ***Test Plan; Environmental Needs***

The test environment is a description of the environment in which the test is to be executed. It is important to be as specific as possible in order to get the right test environment established at the right time (and at the right cost).

## *Test Plan; Responsibilities*

In this section we must describe who is responsible for what. The distribution of testing roles or tasks on organizational units or named people can be shown in a responsibility distribution matrix (RDM). This is a simple two-dimensional matrix or table. On one axis we have organizational units or people, on the other axis we have roles or tasks. In the cross-field we can indicate the type of involvement an organizational unit has for the role.

A completed responsibility distribution matrix might look like this.

	1	2	3	4	5	6
Test leader	R	C	I	I	I	I
Test department	C	R	R	P	P	R
Quality assurance	C	C	R	-	-	I
Sales/marketing	C	C	C	-	-	P
The customer	C	C	C	-	R	P
Method department	I	I	P	R	-	-
<b>Responsible</b>	<b>Performing</b>	<b>Consulted</b>	<b>Informed</b>			

Where:

1. Test management
  2. Test analysis  
and design
  3. Test environment
  4. Test tools
  5. Test data
  6. Test execution

## *Test Plan; Staffing and Training Needs*

The necessary staff to fulfill the roles and take on the responsibilities must be determined and described here.

Each of the roles requires a number of specific skills. If these skills are not available in the people you have at your disposal, you must describe any training needs here. The training should then be part of the activities to put in the schedule.

## *Test Plan: Schedule*

In the scheduling, the tasks, the staffing, and the estimates are brought together and transformed into a schedule. Risk analysis may be used to prioritize the testing for the scheduling: the higher the risk, the more time for testing and the earlier the scheduled start of the testing task.



Scheduling testing is just like any other project scheduling. The result may be presented graphically, typically as Gantt diagrams.

### ***Test Plan; Risks and Contingencies***

This is the management of the risks specifically connected to the task of testing itself, not to the object under test.

The risks to consider here are hence:

- ▶ Project risks—What can jeopardize the plan
- ▶ Process risks—What can jeopardize the best possible performance of the tasks

The risks must be identified, analyzed, mitigated as appropriate, and followed up like any other risk management task.

Risk management is discussed in Section 3.5.



The *approvals* are the sign-off on the plan by the relevant stakeholders.

#### **3.2.3.4 Scheduling Test Planning**

Planning is important and planning takes time.

If you fail to plan—you plan to fail!



It is important to plan activities rather than just jump headfirst into action. The work on the planning provides a deeper understanding of the task at hand, and it is much easier to change something you have written down or sketched out on a piece of paper, than something that has already taken place in the real world.

Because planning takes time and because it is important, it should be planned so that it can start as early as possible. Take your planning seriously, so that you don't end up like this poster painter once did:



The benefits of starting test planning early are many:

- ▶ There is time to do a proper job of planning.
- ▶ There is more time to talk and/or negotiate with stakeholders.
- ▶ Potential problems might be spotted in time to warn all the relevant stakeholders.
- ▶ It is possible to influence the overall project plan.



When you plan you have to keep in mind that a plan needs to be SMART:

- Specific—Make it clear what the scope is
- Measurable—Make it possible to determine if the plan still holds at any time
- Accepted—Make every stakeholder agree to his or her involvement
- Relevant—Make references to additional information; don't copy it
- Time-specific—Provide dates



The test plan should be reviewed and approved by all stakeholders to ensure their commitment. A plan is invalid without commitment from the contributors.

Remember that *a plan is just a plan*; it is not unchangeable once written. A plan must be a living document that should constantly be updated to reflect the changes in the real world. Contrary to what many people think it is not a virtue to keep to a plan at any cost—the virtue lies in getting the plan to align with the real world. No matter how hard you try, you are not able to see what is going to happen in the future.

You should always plan *The New Yorker way*: Adjust the detailing of the planning with the visibility at any given moment. When close to an activity provide many details; for activities further away provide fewer details.

As the time for the execution of activities approaches, more details can be provided, and the necessary adjustments done.

All this takes time and it should not be “invisible” work (i.e., work that is not scheduled reported anywhere). The same in fact holds true for the monitoring activities and for the test reporting.

### 3.2.3.5 Test Report

The purpose of test reporting is to summarize the results and provide evaluations based on these results.

A test report should be issued at the completion of each test level and the end of the entire testing assignment task. The test reports should include analysis of result information to allow management decisions, based on risk, on whether to proceed to the next level of test or to project implementation, or whether more testing is required. Top management may also need information for regularly scheduled project status meetings and at the end of the project in order to adjust policy and strategy.



According to IEEE 829 the test report should contain:

Test report identifier

1. Summary
2. Variances
3. Comprehensiveness assessment
4. Summary of results

#### Test policy

- Definition of testing
- Test process to use
- Test evaluation
- Quality targets
- Test process improvement activities

R

5. Evaluation
6. Summary of activities

#### Approvals

The *test report identifier* is the identification information for the report itself. As for all the other documentation, it could be formed by the name of the report, the organizational affiliation, the version, and the status.

If the report is to be placed under configuration management, the identification should adhere to the organization's standards for configuration management, if there is one.

#### ***Test Report; Summary***

The summary provides an overview of test activities. This section could refer to the test plan. The summary should also include any conclusion. It should be possible to read the summary in isolation and get the main information about the test.

#### ***Test Report; Variances***

The variances to be reported here are all incidents that have happened for any of the items used as a basis for the test. It must also include a summary of what was done and not done with regard to the original plan.

A variance could for example be the issue of a new version of the requirements specification after the approval of the test specification.



#### ***Test Report; Comprehensiveness Assessment***

In this section we report whether we made it or not according to the original (or modified) plan. It should describe which of the planned tests were not performed, if any, and why not.

This is where we must describe how we met the original completion criteria. If they were modified, this is where we explain why.

Any statistically valid conclusions that can be drawn from these analyses could be used to predict the quality level achieved by the tested product. They can also be used to compare with the target level established in the test plans.

#### ***Test Report; Summary of Results***

We must provide an overview of incidents found and incidents solved during the testing.

We can also list findings about which functions are working and which functions are not; or about which risks have been eliminated and which are still outstanding.

The evaluation sums up the expectations versus the actual findings. Any out-of-scope situations should also be documented as should outstanding issues.

We can draw conclusions regarding the quality of software by comparing the planned quality levels with the actual. We can also give recommendations, but it is not the responsibility of the testers to decide whether the test object should be released or not. That is a management decision—project management, product management, or even higher up.



### ***Test Report; Evaluation***

In this section we should give an overall evaluation of the test item, preferably based on a risk analysis of possible outstanding risks related to the release of the item.

The evaluation must be based on the result of the test compared to the completion criteria.

### ***Test Report; Summary of Activities***

Here we must provide an overview of the resource usage for the testing. This could be in terms of time used and other costs such as investments in tools.

The *approvals* here are the approvals of the *test report*, not of the test object.



## **3.3 Test Estimation**

### **3.3.1 General Estimation Principles**

Estimation is a prediction of how much time it takes to perform an activity. It is an approximate calculation or judgment, not something carved in stone. An estimate is typically based on the professional understanding of experienced practitioners.

There are many ways in which to express estimations, but the best way is in hours. In that case we don't get problems with holidays, effective working hours, and so forth. You must never express estimates using dates; dates and estimates are incompatible.



Estimation is input to the scheduling. Only in that activity will we transform the estimated hours into dates.

We can also estimate other elements than just time, for example, number of test cases, number of faults to be found, and number of iterations in the test process needed to fulfill the completion criteria. We may also estimate any other costs, such as hardware and tools.

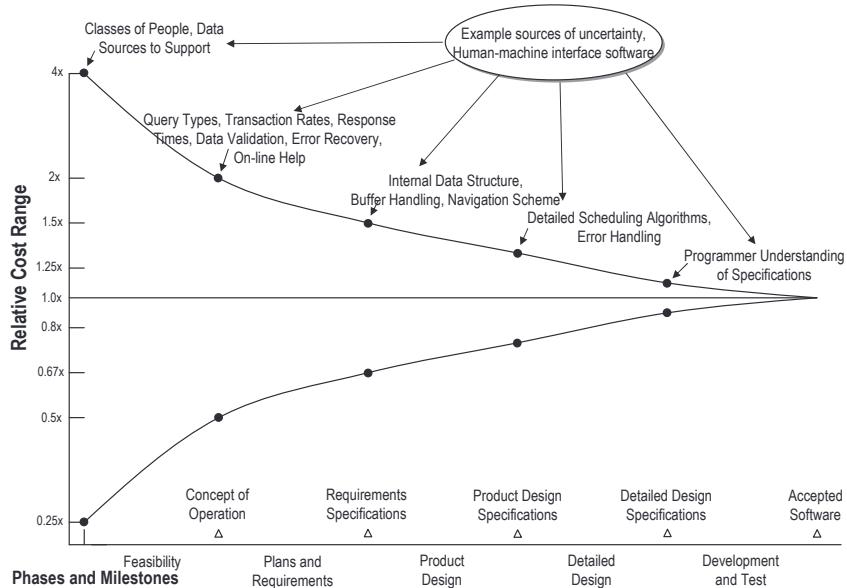
We should always take our estimations seriously. Be honest when you estimate, even though it is often easier to get forgiveness than permission. Keep your original estimates for future reference.

In line with this remember that estimation is not:

- ▶ the most optimistic prediction you can think of
- ▶ equal to the last estimate that was made
- ▶ equal to the last estimate + the delay the customer or the boss is willing to accept
- ▶ equal to a given “correct” answer



Estimates are predictions about the future and predictions are by definition uncertain. The closer we come to the actual result, the less is the uncertainty as illustrated here.



You should always calculate with an uncertainty in every estimate and document this uncertainty with the estimate. Furthermore, estimates should always be accompanied by the rationale or justification for the estimation values along with any assumptions and prerequisites.

### 3.3.2 Test Estimation Principles

Estimating test activities is in many ways like all other estimation in a project. We need to take all tasks, even the smallest and seemingly insignificant, into account.

The time to complete must be estimated for each task defined in the task section, including all the test process activities from test planning to checking for completion.





Even though estimation of testing tasks is in many ways identical to the estimation for any other process, there are also important differences. A test is a success if it detects faults—this is the paradox with which we have to deal.

The test estimation is different from other project estimations, because the number of failures is not known in advance—though it can be estimated as well. The number of necessary iterations before the completion criteria are met is usually not known either. As a rule of thumb, at least three iterations must be reckoned with—one is definitely not enough, unless the completion criterion is a simple execution of all test cases, and independent of the number of outstanding faults and coverage.

Nevertheless, we have to do our best. The estimation must include:

- ▶ Time to produce incident registrations
- ▶ Possible time to wait for fault analysis
- ▶ Possible time to wait for fault correction
- ▶ Time for retest and regression test (minimum three iterations!)

The reason why we have to cater for several iterations is that, well: “*Errare humanum est!*”

When we report incidents and the underlying faults are corrected by development or support staff, not all reported faults are actually corrected. Furthermore, fault correction introduces new faults, and fault correction unveils existing faults that we could not see before.

Experience in the testing business shows that *50% of the original number of faults remains after correction*. These are distributed like this:



Remaining faults after correction	20%
Unveiled faults after correction	10%
New faults after correction	20%

So if we report 100 faults, we have  $20 + 20 + 10 = 50$  faults to report in the next iteration,  $10 + 10 + 5 = 25$  faults in the third, and  $5 + 5 + 2 = 12$  in the forth.

**Ex.**

These are general experience numbers. It is important that you collect your own metrics!

### 3.3.3 The Estimation Process

Estimation is a process like anything else we do. You should of course use your organization’s standard process for estimation, if there is one. Otherwise, you can adapt an estimation procedure like the generic one described here.

1. Define the purpose of the estimation—Is this estimation the first approach, for a proposal, or for detailed planning?
2. Plan the estimating task—Estimation is not a two-minute task; set sufficient time aside for it.
3. Write down the basis for the estimation—Here the scope and the size of the work are determined, and all factors that may influence the estimates are registered. This includes factors related to the nature of the processes we are working by, the nature of the project we are working in, the people we are working with, and any risks we are facing.
4. Break down the work—This is the work breakdown (i.e., the listing of all the tasks to estimate). Do this as well as possible in relation to the purpose.
5. Estimate—Use more than one technique as appropriate.
6. Compare with reality and reestimate—This is the ongoing monitoring and control of how the work that we have estimated is actually going.



### 3.3.4 Estimation Techniques

The following estimation techniques are the most used and an expression of the best practice within estimation.

- ▶ FIA (finger in the air) or best guess
- ▶ Experience-based estimation
- ▶ Analogies and experts
- ▶ Delphi technique
- ▶ Three-point estimation (successive calculation)
- ▶ Model-based estimation
- ▶ Function points
- ▶ Test points
- ▶ Percentage distribution



#### 3.3.4.1 Estimation; Best Guess (FIA)

This technique is more or less pure guesswork, but it will always be based on some sort of experience and a number of (unconscious) assumptions. The technique is very widespread, but since it is based on your gut feeling it is bound to be inaccurate. It is often not repeatable, and it is not always trusted.

The uncertainty contingency is probably around 200%–400% for estimates based on best guess. We can do better than that.

### 3.3.4.2 Estimation; Analogies and Experts

In the analogy techniques you base your estimate on something you have experienced before.

**Ex.**

For example: "This looks very much like the system I tested in my previous job. That took us three months, and we were four people. This is slightly smaller and we are five people—so I guess this will take two months to complete."

If you have participated in a testing project that is comparable to the one you are estimating, you might use that as a baseline to do your estimation.

Analogies may also be based on metrics collected from previous tests. We may estimate the number of iterations of the test based on recent records of comparable test efforts. We can calculate the average effort required per test on a previous test effort and multiply by the number of tests estimated for this test effort.

Experts, in the estimation context, know what they are talking about and have relevant knowledge. It is almost always possible to find experts somewhere in the organization.

If experts on this kind of testing are available, then by all means make use of them. They have been there before, so they know what they are talking about.

### 3.3.4.3 Estimation; Delphi Technique

This is a simple technique that has proved remarkably resilient even in highly complex situations.

You must appoint an estimation group as appropriate. This can be stakeholders and/or experts in the tasks to estimate.

The steps in this estimation process are:



- ▶ Each member of the group gives an estimate.
- ▶ The group is informed about the average and distribution of the estimates.
- ▶ Those giving estimates in the lower quartile and in the upper quartile are asked to tell the rest of the group why their estimates were as they were.
- ▶ The group estimates again—this time taking the previous result and the provided arguments for the "extreme" estimates into account.
- ▶ This may continue two, three, four, or more times until the variation in the estimates is sufficiently small.

Usually the average of the estimations does not change much, but the variation is rapidly decreased. This gives confidence in the final estimation result.

The Delphi techniques can be used in many ways. The people taking part can be in the same room, but they may also be continents apart and the technique used via e-mail.

The technique can be combined with other techniques. Most often the participants give their initial estimates based on experience and/or they are experts in a specific area. The initial estimates may also be obtained using some of the other estimation techniques to make them even more trustworthy.

### 3.3.4.4 Estimation; Three-Point Estimation

Three-point estimation is a statistical calculation of the probability of finishing within a given time. The technique is useful for quantifying uncertainty to the estimate. The technique is also called successive calculation because tasks are broken down and the estimates successively calculated until the variance is within acceptable limits.

Three point estimation is based on three estimates:

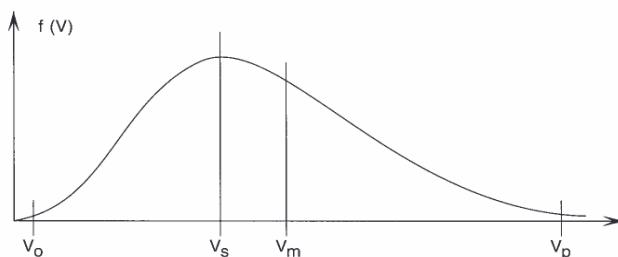
- ▶ The most optimistic time (ideal conditions)
- ▶ The most likely time (if we do business as usual)
- ▶ The most pessimistic time (Murphy is with us all the way)



The three estimates to be used can be provided in a number of ways. One person can be asked to provide all of them, or a group, for example some of the test team members, can take part in the estimation. The estimates can be provided using the Delphi technique or other recognized techniques. High and low values may either be estimated separately (i.e., “what are the best and the worst cases?”) or they may be the highest and the lowest of the individual estimates.

If the “best and worst and most likely” values are used, the estimators should be 99% sure that the actual value will fall between the low and the high values.

From these three estimates it is possible to define the distribution function for the time to finish. It could look like the figure shown where  $V_o$  = most optimistic;  $V_s$  = most likely;  $V_p$  = most pessimistic; and  $V_m$  = mean.



We can use the approximated formula to derive:

$$V_m = (V_o + 3*V_s + V_p) / 5$$

$$S = (V_p - V_o) / 5 \text{ (the standard deviation)}$$

Based on this distribution we can calculate the time needed for any probability of finishing the task we want, by using the appropriate formula.

**Ex.**

For the 5% interval the formulas are:

$$5\%: V_m - 2S \quad 95\%: V_m + 2S$$

Let us say that for a testing task we have reckoned:

$$V_o = 70 \text{ hours} \quad V_s = 80 \text{ hours} \quad V_p = 110 \text{ hours}$$

We calculate:

$$V_m = (V_o + 3*V_s + V_p)/5 = (70 + 3*80 + 110)/5 = 84$$

$$S = (V_p - V_o)/5 = (110 - 70)/5 = 8$$

$$\text{The upper value in the 95\% interval} = 84 + 2 * 8 = 100$$

Therefore if we want to be 95% sure that we'll finish in time our estimate for the given task should be 100 hours.

All tasks or a selection of the most critical tasks can be estimated using this technique.

### 3.3.4.5 Estimation; Function Points

This technique is a factor estimation technique initially published by Albrecht in 1979. It has been revised several times, and it now maintained by IFPUG —International Function Points User Group. The group has been permanent since 1992. Version 4.0 of the technique was published in 1994.

The estimation is based on a model of the product, for example, a requirements specification and/or a prototype.

Five aspects of the product are counted from the model:

- ▶ External inputs
- ▶ External outputs
- ▶ External enquiries
- ▶ Internal logical files
- ▶ External interface files

The counts are then multiplied with a weight and the total of the weighted counts is the unadjusted sum. The actual effort in person hours is then calculated with an adjustment factor obtained from previous project data.

It requires some training to be able to count function points correctly. Continuous comparisons of actual time spent with the estimates are essential to get the best possible local adjustment factor.

The disadvantage of using function points is that they require detailed requirements in advance. Many modern systems are specified using use cases, and use cases are incompatible with this technique.

### 3.3.4.6 Estimation; Test Points

In 1999 Martin Pol et al. published a dedicated test estimation technique called test points as part of the TMAP method.

The technique is based on the function point technique, and it provides a unit of measurement for the size of the high-level test (system and acceptance tests) to be executed.

The technique converts function points into test points based on the impact of specific factors that affect tests, such as:

- ▶ Quality requirements
- ▶ The system's size and complexity
- ▶ The quality of the test basis (the document(s) the test is specified toward)
- ▶ The extent to which test tools are used

### 3.3.4.7 Estimation; Percentage Distribution

Unlike all the other techniques discussed here, this technique is a so-called top-down estimation technique. The fundamental idea is that test efforts can be derived from the development effort.

The estimation using this technique starts from an estimate of the total effort for a project. This estimate may be the result of the usage of appropriate estimation techniques at the project management level.

The next step is to use formulas (usually just percentages) to distribute this total effort over defined tasks, including the testing tasks. The formulas are based on empirical data, and they vary widely from organization to organization.

*It is essential that you get your own empirical data and constantly trim it according to experiences gained.*

If you do not have any data you could assume that the total testing effort is 25–30% of the total project effort. The testing effort should then be spread out on the test levels with an appropriate amount for each level.

This example is from Capers Jones *Applied software measurements*. It is for in-house development of administrative systems. The left-hand table shows the distribution of the total effort on overall tasks, including all tests as one task only. The right-hand table shows the distribution of the effort on detailed testing tasks (the terminology is that of Capers Jones.)



Activity	%	All phases	%
Requirements	9.5	Component testing	16
Design	15.5	Independent testing	84
Coding	20		100
Test (all test phases)	27	Independent testing	%
Project management	13	Integration testing	24
Quality assurance	0	System testing	52
Configuration management	3	Acceptance testing	24
Documentation	9		100
Installation and training	3	System testing	%
		Functional system testing	65
		Nonfunctional system testing	35
			100

### 3.3.5 From Estimations to Plan and Back Again

The estimation is done to provide input to the scheduling activity in the project planning.

In the scheduling we bring the estimates for the defined testing tasks together with the people, who are going to be performing the tasks. Based on the start date for the first task and the dependencies between the tasks we can then puzzle the tasks together and calculate the expected finishing date.

Estimations should be in hours. The scheduling provides the dates: dates for when the performance of each of the tasks should begin, and dates for when they are expected to be finished.

When defining the expected finish date for a task we need to take several aspects into account:

- ▶ The start and/or finish dates of others tasks that this task depends on to start, if any
- ▶ The earliest possible start date for the task
- ▶ The general calendar regarding public holidays
- ▶ The pure estimate for the time to finish the task
- ▶ The efficiency of the employee(s) to perform the task—typically 70–80% for a full time assignment
- ▶ The employee(s)'s availability—this should NOT be less than 25%



We should not expect that our estimations are accepted straightaway. Making plans for a development project is a very delicate balance between



resources (including cost), time, and quality of the work to be done. Testing is often on the critical path for a project, and testing estimates are likely to be the subject of negotiations between stakeholders—typically the customer or higher management, the project manager, and the test manager.

*The estimating does not stop with the preparation of the first schedule.* Once the actual testing has started—from the first planning activities and onwards, we need to frequently monitor how realities correspond to the estimates. Based on the new information gathered through the monitoring, we must re-estimate, when the deviations between estimates and reality get too large to stay in control. Only when all the testing activities are completed can we stop the monitoring and re-estimation.



### 3.3.6 Get Your Own Measurements

All estimates are based on experience—maybe very informally (FIA), maybe very formally (like function points). The better the basis for the estimation is, the better the estimation gets. Better estimation means more reliable estimations, and that is what we both, management and customers, want.

In order to get better estimates we need to collect actual data. The more empirical data we have, the better will the estimates be. In general we can say that (almost) any data is better than no data.



We do, however, always need to objectively evaluate the empirical data we have—is it collected from tasks that can be compared with the ones we are dealing with now? When we use the empirical data available, we also have an obligation to contribute to and refine the empirical data on an ongoing basis.

Empirical data for estimation is part of the measurements we are collecting. So we need to chip in to establish a set of simple measurements of time, costs, and size in all projects we participate in. This requires procedure(s) for collection of data to be established and maintained, and training of the (test) managers in these procedure(s).

From an organizational point of view it must be checked that all completed projects collect data, and the usage of these data must, of course, also be enforced.

## 3.4 Test Progress Monitoring and Control

Continuous monitoring of how the test is progressing compared to the plan is absolutely necessary to stay in control. If we don't control the test project, it will control us—and that is not a nice experience.

*You need to collect information about facts, compare these with the estimates, and analyze the findings.* This is needed to minimize divergence from the test plan. If there is any discrepancy you need to take action to keep in control, and you need to inform the stakeholders.



There are a few rules that you must adhere to when you do the follow-up on the actual activities. Follow-up must be guided by:

- ▶ Honesty
- ▶ Visibility
- ▶ Action

First of all you need to be honest, not only when you estimate, but also when you collect information about reality. In the long run you lose integrity and trust if you “tailor” the numbers, or come up with “political” results of the monitoring.

You also need to make the information visible to all stakeholders. Again you lose trust if you hide the truth, be it a positive truth (we are ahead of schedule) or a negative truth (we are behind schedule). Information about progress and findings must be made readily available to the stakeholders in appropriate forms.

The last thing you need to do to stay in control is to take action whenever needed. *It is your duty as test manager to intervene as soon as deviations appear!*



### 3.4.1 Collecting Data



The data to collect during testing should be specified in the approach section in the test plan, based on the requirements outlined in the policy and the strategy.

The concept of metrics and measurements is discussed in Section 1.3.

No matter which data we have planned to collect it is not enough to just collect it. It must be presented and analyzed to be of real value.

### 3.4.2 Presenting the Measurements



Test reports are used to communicate test progress. These reports must be tailored to the different recipients or stakeholders. The immediate stakeholders for test monitoring information are the customer, the project and/or product management (or higher), the test management, and the testers.

The customer and the management above test management need test reports (described below) when the test is completed. The test management needs information on a continuous basis to keep in control. The testers need to be kept informed on progress at a very regular basis—at least daily when the activities are at their peak.

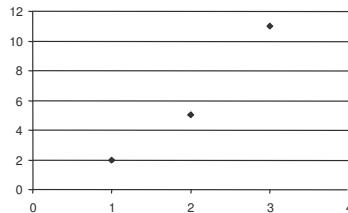
A picture speaks a thousand words. The best way to present progress information for testing is by using graphics. This holds true for all stakeholders, though especially for the testers in the middle of the action. Graphics used in the right way give an immediate overview—or feeling—for the state of the testing.

The flip side of the coin is that graphics can “lie.” You can do it deliberately—which is outside the scope of this book—or you can make it happen accidentally if you are too eager to make your presentation “interesting” and “lively.” The truth is usually boring, but adding decoration does not help.

One of the common mistakes is to use too many dimensions. Most of our information is one-dimensional: the number of something. Many graphs, however, present one-dimensional information in a two- or even three-dimensional way.

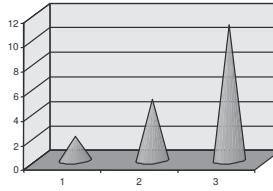
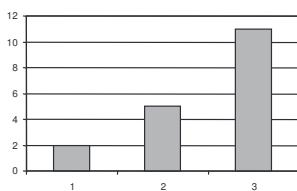
Consider the following information:

Day 1: 2 faults found  
Day 2: 5 faults found  
Day 3: 11 faults found



The simplest way to present this is as shown to the right. See the trend? Yes, that is perfectly clear! Need anything else? Not really.

But all too often we may see exactly the same information presented like this: or, even worse, like this:



Does that add to the understanding? No.

There is a “metric” called the ink-factor. That is defined as the amount of ink used to convey the message compared to the amount of ink used in the graph. You should keep the ink-factor as low as possible.

Also avoid *highlighting* (read: *hiding*) the message in decoration, patterns, shading, or color. A graph that presents the number of failures found each day as the size of the corollas of a line of flowers is perhaps cute, but not professional.

More obvious ways to misinform is by changing the scale across the axis, or by omitting or distorting the context of the information or the way it has been collected.



Sources:  
Tufte and Huff



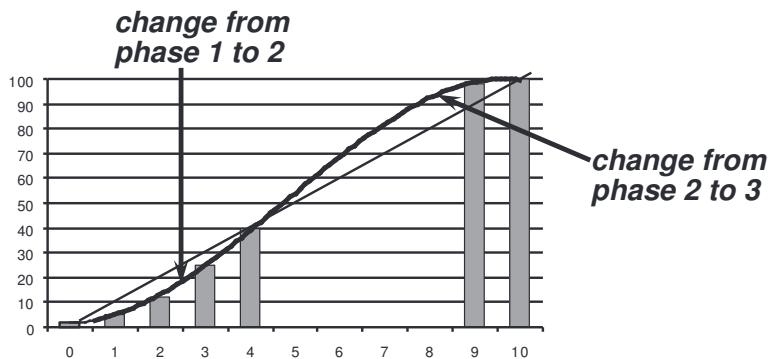


Whichever way you choose to present the information you have collected, it is your responsibility to ensure that the recipients understand and interpret the data correctly. In the following some of the most common and useful ways of presenting test progress information are described.

### 3.4.2.1 S-Curves

The most used, most loved, and most useful way of presenting progress information and controlling what's happening is S-curves. They are named so because of the shape of the wanted curve.

Source:  
Marnie  
Hutcheson,  
Unicom  
Seminar,  
Oct 95.



S-curves can be used for many different metrics. You need two that are related to each other—one is typically time. The other could, for example, be:

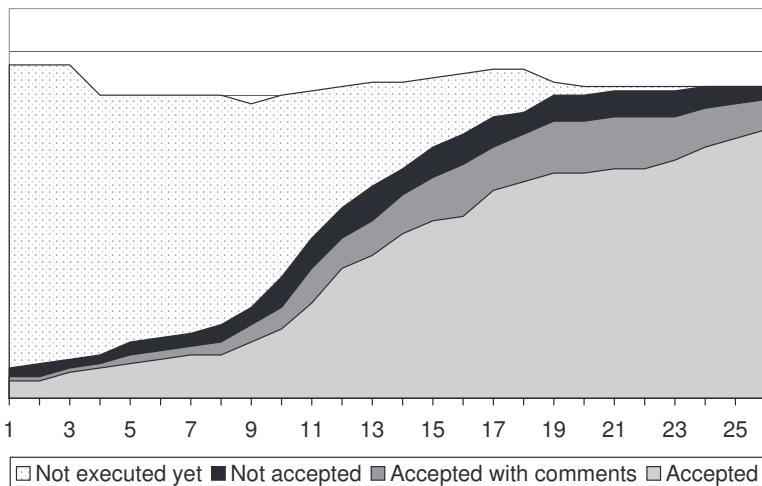
- ▶ Test cases (run, attempted, passed, completed)
- ▶ Incidents (encountered, fixed, retested)

S-curves can give us early warnings of something being wrong. It can also give us reassurance that (so far) things are progressing as planned.

The principle in S-curves is that our work falls in three phases:

- ▶ Phase 1: Slow start—not more than 15–25%  
An initial period of reduced efficiency to allow for testing teams to become familiar with the testing task, for example with the test object, the test environment, and execution and logging practices.
- ▶ Phase 2: Productive phase—55–65%  
After the initial period, the second phase is the period of maximum efficiency.
- ▶ Phase 3: The difficult part—10–25%  
The final phase reflects the need to be able to ease off the work as the testing window nears completion.

The following figure shows how real data are reported as several S-curves in the same graph.



To use an S-curve you need to know what the expected start point (usually 0,0) and the expected end point are. The end point is your estimation of what needs to be achieved by the end time; for example 300 test cases passed after 21 days of testing.

You mark the start point and the end point, and you draw (or get a tool to draw) a third-order polynomial that fits. A straight-line approximation between the two points may do.

As the time goes and you do your work, you plot in your achievements—for example, the sum of test cases passed day by day. Follow the progress to see if it fits the predicted curve. If it does, we are happy!

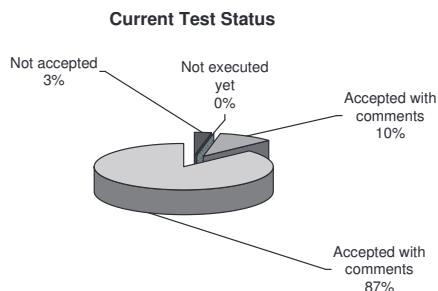
If the upward turn, marking the start of the second phase, comes too late, we are in trouble. But the good news is that we know it and can take action well before the end date! If the curve is rising too fast, we may also be in trouble, and we must investigate what could be wrong. Maybe our test cases are not giving enough failures? Maybe we have run the smallest first and are pushing work in front of us?

### 3.4.2.2 Pie Chart

Pie charts are used to give an overview of the proportions of different aspects relative to each other. Pie charts are perhaps the most used graph in the world.

The graph shown here gives a nice impression of the testing going well.

But think about the inkfactor — maybe the third dimension is not needed to present the information. Maybe it is even disturbing the impression: Should the lightest gray volume be almost nine times as big as the medium gray?



### 3.4.2.3 Check Sheets

Check sheets are a good way to give a quick overview of status. They can be used to show progress compared to plan, for example, for planned test cases, planned test areas, or defined risks.

Check sheets can be presented as colorful graphics or expressed as lists or hierarchies. They are usually easy to produce, and easy to understand. Some organizations make wall-sized check sheets and stick them in the meeting room or the corridor. This way everyone has easy access to the information about the progress of the test.

A few examples of check sheets are shown next.

The first is an extract of the check sheet presented on Systematic's intranet. It is updated every day. Even though the text has been deliberately blurred and the extract is small, it gives an impression of things going well.— no black or light gray fields in the list!

**Ex.**

Status for project: ksdf				
Area	Remain	% Complete	Status	Comment
agfdg	8	56		
gstyk	0	100		
jl_flli	0	100		
dsrahjtdulk	1	80		
ths	2	56		
jdvw	0	100		
yjdtrek	0	100		
	0	87		



The next is a dashboard suggested by James Bach:

**Ex.**

Area	Test effort	Coverage planned	Coverage achieved	Quality	Comments
Start up	High	>80%	27%	😊	ER 52
Discount	Low	>40% <70%	53%	😊	
Pricing	<b>Blocked</b>	>40% <70%	14%	😢	<b>ER 86</b>

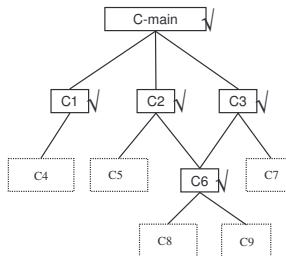
James Bach's recommendations for the presentation are: draw it on the wall, make it huge, and update it every day.

**Ex.**

The last example here is a tiny extract of a hierarchical check sheet showing the progress of a component test for a system.

The marked components have been successfully component-tested and are ready for integration.

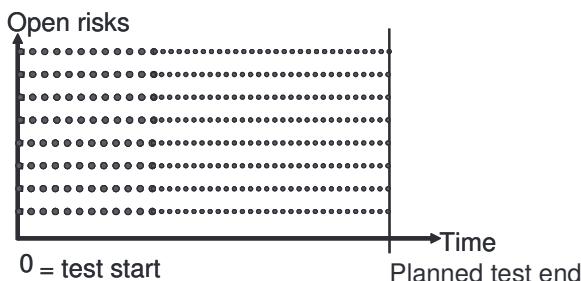
The integration testing has not yet started—no interfaces are marked as having been successfully tested. It is, however, easy to see which interfaces we have to test.



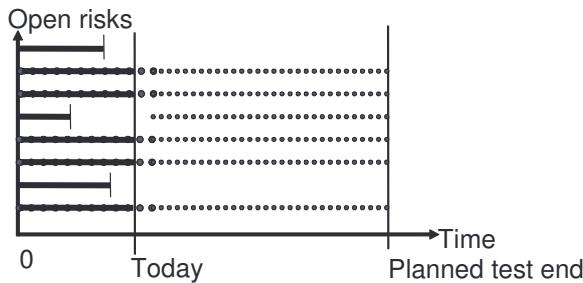
#### 3.4.2.4 Risk-Based Reporting

If our test approach is based on identified and analyzed risks it is appropriate to report on the test progress in terms of these risks.

The purpose of this test is to eliminate the risk, so the reporting must be in terms of eliminated risks. The open risks at the test start could be illustrated like this:



At any point in time we must make it possible to see from the updated progress graph which risks are still open, if any. The risks with the small vertical line across them are eliminated and hence no longer present any threat to the system!



### 3.4.2.5 Statistical Reporting

The way a process is performed is different from project to project and over time, because processes are performed by people, not machines.

Statistics is the science of patterns in a variable world. We can say that statistics make the invisible visible. This means that statistical methods can be used to help us:

- ▶ Understand the past
- ▶ Control the present
- ▶ Predict the future

Statistics also include handling of “fortuitousness,” that is, happenings that are out of the ordinary.

When we have to deal with many happenings assumed to be “alike,” we need to find out what “alike” means. To do that we must find out what the norm is, and what variances are allowed to still call things “normal.”

**Ex.**

Norm and variation vary. In our family the norm is that we are friendly and talk to each other in nice, calm tones of voices. I, however, have a short temper, and sometimes raise my voice without anything being really out of the normal. If, on the other hand, I keep quiet, then my mood is not within the norm. My husband is different: If he raises his voice just a little bit, he is sure to be in a very bad mood.

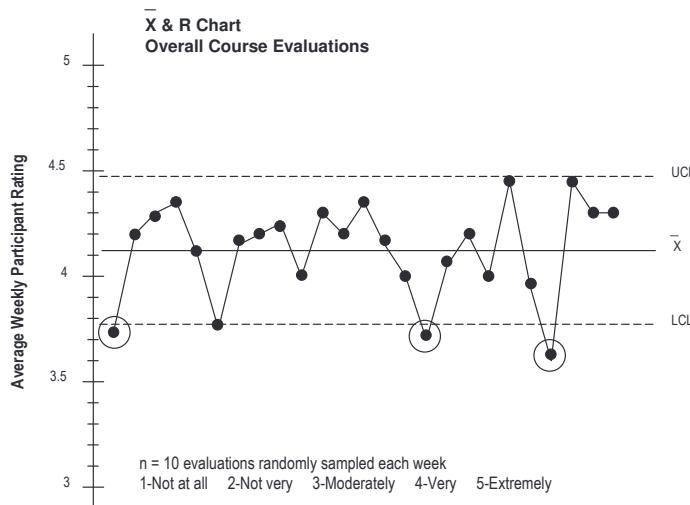
The norm in statistics can be calculated in three much used ways, namely: mean—the arithmetic average; median—the value that splits the group in the

middle; and modus—the most frequent value.

But how far from the “normal” value, can a given value be and still be considered within the norm? This can be seen in a control sheet.

An example of a control sheet is shown here. The values are ratings for a course. Every week 10 evaluations are randomly sampled and the average is plotted in the graphs. The graph shows the upper and lower control levels (UCL and LCL) for the series of ratings.

**Ex.**



The values here are indicators for the course performance. We can choose other values to be indicator values for our processes, if we want to control how they are performed.

An indication value may, for example, be the average time per test case it takes to produce a test specification.

**Ex.**

When we examine the control sheet we must be looking for warnings of something being out of the borders of the norm. Such warnings may, for example, be:

- ▶ One value outside either CL
- ▶ Two out of three values on the same side
- ▶ Six values in a row either up or down
- ▶ Fourteen values in a row alternately up and down

Many tools can assist in the necessary statistical calculations. The use of control sheets and statistics is rather advanced process control—belonging to maturity level 4—and we will not go into further depth here.

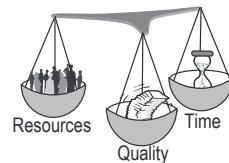
### 3.4.3 Stay in Control

Sometimes we need to take action to stay in control.

Keeping the triangle of test quality in mind, you have three aspects you can change—and you must change at least two at the time.

The aspects are:

- ▶ The resources for the task
- ▶ The time for the task
- ▶ The quality of the work to be performed



Usually when things are getting out of control it is because we are behind schedule or because our time frame has been squeezed. To compensate for this we must (try to) obtain additional resources and/or change the quality of the testing. The latter can be done by changing the test completion criteria and/or changing the amount or depths of the tests to be performed.

Any change you make must be reflected in the plan. The plan must be updated with the new decisions based on the new information. The new plan must be reviewed and approved, just like the first one.

Remember that *it is not a virtue to comply with the plan at any cost—the virtue lies in the plan complying with reality*.

No matter which way the progress is measured and presented, the test manager must analyze the measurements. If something seems to be going the adverse way, further analysis must be made to determine what may be wrong.



**Ex.**

Examples of things not being as they should be are:

- ▶ The delivered software is not ready for test
- ▶ The easy test cases have been run first
- ▶ The test cases are not sufficiently specified
- ▶ The test case does not give the right coverage
- ▶ General faults have wide effects
- ▶ Fault correction is too slow
- ▶ Fault correction is not sufficiently effective

Based on this analysis the test manager must identify what can be done to remedy or mitigate the problems.

Possible actions may, for example, be:

- ▶ Tighten entry criteria
- ▶ Cancel the project
- ▶ Improve the test specifications
- ▶ Improve the incident reporting
- ▶ Perform a new risk analysis and replan
- ▶ Do more regression testing

**Ex.**

The important message to the test manager is that he or she must intervene as soon as deviations appear! Or in other words:

*If you do not control the test, it will control you!*

To sum up we can say that as test managers we must set out the destination and plan how to get there; collect data as we go along; analyze data to obtain information; and act on the information and change destination and plan as appropriate.



## 3.5 Testing and Risk

The golden rule of testing is:

**Always test so that whenever you have to stop  
you have done the best possible test.**

Everybody with some understanding of requirements and tests will know that a requirement like this cannot be verified. What does it mean: the best possible test? It is not immediately measurable.

What is the best possible test then? The answer to that is: It depends!

*The best possible test depends on the risk associated with having defects left in the product when it is released to the customer!*

The best possible test is determined by the risks we are facing and the risks we are willing to run. Obtaining consensus from stakeholders on the most important risks to cover is essential.



### 3.5.1 Introduction to Risk-Based Testing

We have to live with the fact that it is impossible to test everything. Testing is sample control. There is a risk involved in all sample control: the risk of overlooking defects in the areas we are not testing.

### 3.5.1.1 Risk Definition

A risk is defined as: "The possibility of realizing an unwanted negative consequence of an incident."

Alternatively, a risk may be defined as: *'A problem that has not materialized yet and possibly never will.'*



There are two important points in these definitions:

- ▶ A risk entails something negative
- ▶ A risk may or may not happen—we don't know

A risk therefore has two aspects:

- ▶ Effect (impact—consequence)
- ▶ Probability (likelihood—frequency)

It is not a risk (to us), if there is no effect of an event that might happen. We can therefore ignore it even if the probability is high.



There is a probability that there are defects in the new version of our database management system, but that will have no effect on the quality of our product if it happens, because it is not used in our system.

It is not a risk if there is no (or an extremely small) probability that an event will happen, even if the effect would be extremely big, if it did. We can therefore ignore that as well.



There is no (detectable) risk of our department closing down, because we have lots of orders, and are making good money, and both management and employees like their jobs. If we did close down the effect on the project would be pretty bad, if not disastrous.

It is not a risk either if the probability of an event with a negative effect is 100%. In this case we have a real problem on our hands, and we will have to deal with that in our planning.



It is a problem—not a risk—that we will have to do without one of our test experts because she has found another job and is leaving in three months.



The two aspects of risk can be combined in

$$\text{Risk level} = \text{probability} \times \text{effect}$$

From this it is quite clear that if we have no probability or no effect we have no risk.

The risks that do have a risk exposure greater than zero are the risks we have to deal with.

### 3.5.1.2 Risk Types

It is quite common to treat all the risks we can think of in connection with a development project in one big bundle. This can be quite overwhelming.

It is therefore a very good idea to take a closer look at the risks and divide them into classes, corresponding to where they may hit, or what they are threatening.

Risks hit in different places, namely;

- ▶ The business
- ▶ The processes
- ▶ The project
- ▶ The product



The risks threatening the product are the testers' main concern. This is where we can make a difference.



The business risks are things threatening the entire company or organization from a "staying-in-business" point of view. This is out of the scope of this book, and will not be discussed further.

Process risks are related to the processes and/or the way work is performed. It is also out of the scope of this book, but will be briefly discussed because knowledge about processes is indispensable in a modern development organization.

Process risk threatens the effectiveness and efficiency with which we work on an assignment. Process risks may be originated in:

- ▶ Missing process(es)
- ▶ The organization's lack of knowledge about the processes
- ▶ Inadequate processes
- ▶ Inconsistencies between processes
- ▶ Unsuitable processes
- ▶ Lack of support in the form of templates and techniques



Process risks jeopardize the way the work in the project is being performed. These risks should be the concern of the project manager and those responsible for the processes in the organization.

Process risks may influence business, as well as project and product risks.

### ***Project Risks***

Project risks are related to the project and the successful completion of the project.

A project consists of a number of activities and phases from requirements development to the final acceptance test. These activities are supported by activities like quality assurance, configuration management, and project management.

All the activities in a project are estimated, get allocated resources, and are scheduled. As the project progresses the activities are monitored according to the plan.

Risks concerning the project may be originated in:

- ▶ People assigned to the project (e.g., their availability, adequate skills and knowledge, and personalities)
- ▶ Time
- ▶ Money
- ▶ Development and test environment, including tools
- ▶ External interfaces
- ▶ Customer/supplier relationships

Project risks jeopardize the project's progress and successful completion according to the plan.

**Ex.**

Examples of project risks are:

- ▶ The necessary analysts are not available when the requirements development is expected to start
- ▶ Two of the senior designers are not on speaking terms and useful information exchange between them is not done—this causes the design phase to take longer than expected
- ▶ The complexity of the user interface has been underestimated
- ▶ The testers are not adequately trained in testing techniques, so testing requires more resources than expected
- ▶ The integration is more time-consuming than expected
- ▶ The access to external data is not possible with the technique chosen in the design

The project risks are the main concern of the project manager and higher management.

The test manager is concerned with the project risks related to the test project as it is specified in the test plan documents.

### ***Product Risks***

Product risks are related to the final product. They are the risks of defects remaining in the product when it is delivered.

We want to deliver the required quality and reliability. This cannot be tested into the product at the end of the development, but must be worked into the product through the work products produced during development and in the implementation of the components.

Product risks may be originated in:

- ▶ Functional and nonfunctional requirements
- ▶ Missing requirements
- ▶ Ambiguous requirements
- ▶ Misunderstood requirements
- ▶ Requirements on which stakeholders do not agree

More than 50% of all defects in products can be traced back to defects in the requirements.



- ▶ Design
- ▶ Not traceable to requirement
- ▶ Incorrect
- ▶ Incomplete (too superficial)
- ▶ Coding
- ▶ Testing
- ▶ Not traceable
- ▶ Inadequate

Product risks jeopardize customer satisfaction and maybe even the customer's life and livelihood.

Product risks may be related to different requirement types, such as functionality, safety, and security and political and technical factors.

Examples of product risks are:

- ▶ A small, but important functionality has been overlooked in the requirements and is therefore not implemented
- ▶ A calculation of discounts is wrongly implemented, and the customer may lose a lot of money
- ▶ The instrument may reset to default values if it is dropped on the floor
- ▶ It is possible to print a report of confidential customer information through a loophole in the reporting facility
- ▶ The installation procedure is difficult to follow

**Ex.**

These risks are the main concern of the testers, since testing may mitigate the risks. The test strategy for a product should be based on the product risks that have been identified and analyzed.

Project risks and product risks can influence and be the cause of each other. A project risk may cause a product risk, and a product risk may cause a project risk.

**Ex.**

If a project risk results in time being cut from component testing, this may cause the product risk of defects remaining in the components that are not tested or not tested sufficiently. This may further cause the project risk that there is not sufficient time to perform a proper system test because too many trivial failures are encountered in the system test.

### 3.5.1.3 Testing and Risk Management

Testing and management of risks should be tightly interwoven as they support each other. Testing can be based on the results of risk analysis, and test results can give valuable feedback to support continuous risk analysis.

The result of a product risk analysis can be used in test planning to make the test as effective as possible. It can be used to target the testing effort, since the different types of testing are most effective for different risks.

**Ex.**

Component testing is most effective for testing where the product risk exposure related to complex calculations is highest.

The risk analysis results can also be used to prioritize and distribute the test effort. The areas with the risk exposure should be planned to be tested first and given the most time and other resources.

Finally the product risk analysis can be used to qualify the testing already done. If test effort is related to risks it should be possible to report on dissolved and remaining risks at any time.

Testing can, as mentioned earlier, dissolve or mitigate the product risks. The probability of sending a product with defects out to the customers is reduced by the testing finding failures and the subsequent correction of the defects.

Testing can also mitigate project risks if an appropriate test strategy is applied, especially if testing is started early.

Even process risks may be reduced by analyzing failure reports and taking appropriate process improvement initiatives.

### 3.5.2 Risk Management

Risk management consists of the following activities:

- ▶ Risk identification
- ▶ Risk analysis
- ▶ Risk mitigation
- ▶ Risk follow-up



In risk identification we are finding out what may happen to threaten the process, the project, or, in this particular context, the customer satisfaction for the product.

The identified risks are evaluated in the risk analysis and ordered relatively to each other. The analysis means that we assign probability and effect to the risks. Based on this we can determine the risk exposure and hence which risk is the worst and how the others relate to that.

One of the points in risk management is to use the results of the analysis to mitigate the risks. Actions can be planned to lower the probability and/or the effect of the risks. In this context of product risks and testing, test activities can be planned to mitigate the risks by lowering the probability of having remaining defects in the product when it is released. The more defects we can remove from the product as a result of the testing, the more the probability falls.

Contingency planning is a part of classic risk management, but this is not relevant for product risks in relation to testing. Testing is concerned with lowering the probability of remaining defects for the defects that remain; support and maintenance must be prepared to provide work-arounds and/or corrections and updates.

Risk identification, analysis, and mitigation must not be a one-time activity. It is necessary to follow-up on the risks as testing progresses. The results of the testing activities provide input to continuous risk management.

Information about the failure frequency over time can be used to assess if the probability of a risk is falling or rising.



There are many stakeholders in a development project, and they come from different places and have different view points, also on risks.

The following table shows examples of stakeholders or stakeholder representatives for a number of aspects of a development project.

Aspect	Stakeholder representatives
The business	Product line manager Business analysts Marketing personnel Finance executives
Future users	Future users Users of existing system Users representing different types of user groups
User representations	Marketing personnel Salespeople Employees from relevant support function
Technical, development	Analysts, designers, programmers, testers People responsible for manufacturing People responsible for operation People responsible for configuration management People responsible for quality assurance
Technical, product	Experts in and people responsible for usability, security reliability, performance, portability

All stakeholders identified for a project should be involved in the risk management activities.

### 3.5.2.1 Risk Identification

Risk identification is finding out where things can go wrong and what can go wrong, and writing it down to form the basis for risk analysis.

Risks are found in areas where the fulfillment of expectations may be threatened (i.e., where customer satisfaction is jeopardized). *Satisfying expectations is the way to success!*

All stakeholders have expectations towards the product, but we are most concerned with the expectations of the customer. The customer is the one to order the product, to pay for it, and to take advantage of it, the latter possibly through end users.

In the ideal world the customer's expectations are expressed in user or product requirements. These requirements are transformed into design, and the requirements are fulfilled in the code and maybe also in other subsystems, being integrated into the final products.

In less ideal worlds expectations may be derived from other sources.

Risks are not always evident. Even when we work with experienced and knowledgeable stakeholders it can be efficient to use a risk identification technique.



Customer satisfaction

Useful techniques are:

- ▶ Lessons learned
- ▶ Checklists
- ▶ Risk workshops
- ▶ Brainstorms
- ▶ Expert interviews
- ▶ Independent risk assessments



Techniques may be mixed to be even more efficient.

### 3.5.2.2 Lessons Learned and Checklists

Lessons learned and checklists are closely related. A checklist is a list of generic risks formed and maintained by experience (i.e., lessons learned from previous projects).

*Risk checklists are valuable assets in an organization* and should always be treated as such.



One or more product risk checklists should be kept in the organization, depending on the diversity of the nature of the projects performed in the organization.

A product risk checklist could be structured as the requirements specification or the design specification is structured, or both.

The checklists used by the pilots before takeoff are long and must be run through very carefully before every takeoff. A pilot was once hurried on by a busy business man and asked to drop the checklists and get going. He carried on with his work as he answered: "These checklists are written in blood!"



### 3.5.2.3 Risk Workshops

Workshops are an effective way to identify risks. There are no strict rules as to how a workshop should be conducted, but a few guidelines can be given.

As many stakeholders as possible should be involved, though the number of participants should not exceed 10–12 in order to give everybody a chance to talk within reasonable intervals.

Risk workshops can get emotional, and a neutral facilitator—somebody who is not by any account a stakeholder—should be present to guide the discussions. *Encourage discussions and new ideas, but avoid conflicts.*

Make sure that all participants agree that the objective has been reached and that it is clear how work can proceed after the workshop.



### 3.5.2.4 Brainstorming

A brainstorm (in this context of risk identification) is an informal session with the purpose of identifying possible risks connected with the product when it is released.



The only rule that should apply during a brainstorm *is that no possible risk must be commented on* in any way by the participants. Ideas should be allowed to flow freely, the rationale being that even the most seemingly stupid, silly, or strange thought may be the inspiration for valuable potential risks.

A brainstorm must have a facilitator who may act as a catalyst if ideas do not flow freely. At the end of the session the facilitator must make sure that whatever is being brought forward as possible risks is expressed as risk and documented.

The stakeholders to be involved in this technique may be any type of stakeholders.

### 3.5.2.5 Expert Interviews



Interviews may be conducted as individual interviews or as group interviews. *An interview is not as easy to conduct* as many people think. It requires specific skills and thorough preparation to get as much information as possible from an interview.

First of all, an interview is not like an ordinary conversation. People in an interview have different roles (i.e., the interviewer and the interviewee(s)), and they may have a number of expectations and prejudices related to these roles. Interviews must be prepared. The interviewer must, for example, make sure that the right people are being interviewed and the right information is gathered. A list of questions or a framework for the course of the interview must be prepared.



Ample notes must be taken and/or the interview can be recorded with the permission of the interviewee(s). *The interviewer must extract a list of possible risks from the interview* and get agreement from all the participants.

### 3.5.2.6 Independent Risk Assessments

In cases where conflicts are threatening external consultants may be called in to identify risks. External consultants could also be used if time is short or if specific expertise not present with the immediate stakeholders is required.

The external consultants identify risks and usually perform or facilitate the risk analysis.

The consultants may be external to the project organization or third-party consultants entirely external to the developing organization.

### 3.5.3 Risk Analysis

Risk analysis is the study of the identified risks. One thing is to identify and list the risks; another is to put them into perspective relative to each other. This is what the analysis of the risks helps us do.

*The analysis must be performed by all the appropriate stakeholders,* because they all have different perspectives and the risk analysis aims at providing a common and agreed perspective. Experts may be called in to contribute if adequate expertise cannot be found among the immediate stakeholders.

Risk analysis can be performed more or less rigorously, but it should always be taken seriously.



#### 3.5.3.1 Risk Template

A risk template or a risk register is a very useful tool in risk management. It can be used to support risk analysis and risk mitigation.

Risk templates can be held in office tools, for example, spreadsheets that support calculations.

A risk template should include:

- ▶ Risk identification (e.g., number or title)
- ▶ Risk description
- ▶ Probability
- ▶ Effect
- ▶ Exposure
- ▶ Test priority
- ▶ Mitigation action = test type
- ▶ Dependencies and assumptions



#### 3.5.3.2 Perception of Risks

The performance of risk analysis can be more or less objective. In fact *most risk analysis is based on perceptions*; it is usually not possible to determine risk probability and effect totally objectively. There is an element of prediction in risk analysis since we have to do with something that has not happened and maybe never will. There are usually very few trusted measurements applicable to identified risks.



Perceptions are personal and different people have different “pain thresholds.” Just look around you: Some people use their holidays to explore new places; others always spend their holidays at the same place. In connection with process improvements we sometimes say that if it blows hard some people build shelters; others build windmills.

**“If you do that, I’ll never see you again!!!”**  
**“Is that a promise or a threat?”**



People in different professions may also have different view points on risks, partly because people choose jobs according to their personalities, partly because job-related experiences influence their perception of different risks.

The following descriptions of job-related risk perceptions are of course gross generalizations, but they can be used as guidelines in understanding different viewpoints on “the same risks.” The descriptions encompass:



- ▶ Project managers
- ▶ Developers
- ▶ Testers
- ▶ End users

*Project managers* are often under time pressure; they are used to compromises. They know that even though things may look dark, the world usually keeps standing.

*Developers*, that is analysts, designers, and programmers, are proud of their work, and they know how it was done. They have really done their best, and they are usually reluctant to accept that there may still be defects left in there.

*Testers* often have a pessimistic view on work products, product components, and products. We remember previous experiences where we received objects for testing and got far more failures than we expected.

The *end users* are, despite what we might think, usually highly failure-tolerant. They also tend to remember previous experiences, but what they remember is that even though the system failed, they found a way around it or another way of doing their work. End users use our product as a tool in their job, and nothing more. If it does not help them, they'll find another way of using the tool, find another tool, or just live with it.



In risk analysis we must encourage communication and understanding between stakeholders. *Stakeholders need to be able to, if not agree with then at least be aware of and accept others' points of view.* If need be, stakeholders will have to compromise or use composite analysis. This is explained next.

### 3.5.3.3 Scales for Risk Analysis

The analysis of risks uses metrics for probability and effect. For all work with metrics it is mandatory to use agreed and understood scales. This is therefore also the case in risk analysis.

We can work with two different kinds of scale, namely:

- ▶ Qualitative
- ▶ Quantitative

In a qualitative scale we work with feelings or assessments.

For effect we could use

bad—worse—worst

For probability this could be expressed as

not likely—likely—very likely

**Ex.**

In a quantitative scale, on the other hand, we work with exact measures or numbers.

For effect we could use actual cost in \$ or Kr. or €.

For probability this could be expressed as

$\leq 10\%$ ,  $> 10\% \& \leq 50\%$ ,  $> 50\% \& \leq 80\%$ ,  $> 80\%$

**Ex.**

Whichever way to do it, we must define and agree on scales for both probability and effect before we start the risk analysis, that is before we assign metrics to the probability of the identified risks actually materializing, and metrics for the effects if they do.

### 3.5.3.4 Effect

The effect is the impact or consequences of a risk if (when?) it occurs. The first thing we have to do is agree on a scale for the effect.

The obvious *quantitative scale* for the effect is the actual cost imparted by a failure occurring out in the field. The actual cost can be measured in any agreed currency (\$, €, Kr.). This is an open scale; in theory there is no limit to actual cost.

It can be very difficult to assess what the actual cost in real money might be. On the other hand it can be quite an eye-opener to sit down and consider all the sources of extra cost associated with a failure.

Expenses may for example be considered for:

- ▶ Time for the end user to realize that something is wrong
- ▶ Time to report the incident to first-line support
- ▶ Time for first-line support to understand the report and try to help
- ▶ Time for any double or extra work to be performed by the end users
- ▶ Loss of production because the system is down or malfunctioning
- ▶ Time for escalation to secondhand support
- ▶ Time for secondhand support to try to help
- ▶ Time to investigate the failure and decide what to do about it
- ▶ Time for finding the defect(s)
- ▶ Time for corrections to be implemented and tested in all affected objects

**Ex.**

- ▶ Time for retesting and regression testing
- ▶ Time to reinstall the new version
- ▶ Time to update what has been done by other means while the system was unavailable or malfunctioning

These are all examples of time spent in connection with a failure. There may also be costs associated with, for example, renting or replacing parts of the system or the entire system.



Furthermore there may be an effect in the form of indirect losses from, for example, people getting hurt, the environment being destroyed, or the company getting an adverse reputation or losing trustworthiness.

Failures have been known to cost lives or to put companies out of the market completely. Fortunately, it is usually not that bad, but still the effects of failures can be significant.

Another way to measure effect is by using a qualitative scale. Such a scale could, for example, be expressed as shown in the following table.



Inspired  
by Paul Gerrard

<b>Effect</b>	<b>Description</b>	<b>Score</b>
Critical	Goals cannot be achieved	6
High	Goals will be jeopardized	5
Above middle	Goals will be significantly affected	4
Below middle	Goals will be affected	3
Low	Goals will be slightly affected	2
Negligible	Goals will be barely noticeably affected	1

In the table there is a column for a mnemonic for the effect, a column describing the effect more precisely and a column for the actual score.

Using a numeric score makes it possible to calculate the risk exposure even when a qualitative scale is used for the effect.

Despite the above example it can be useful to define a scale with an even number of scores. This can mitigate the effect of some people having a tendency for choosing the middle value if they are not sure what to score or can't be bothered to think deeper about their opinion. A scale with an even number of scores does not have a middle value and the stakeholders will have to decide if they want to score over the middle or under.



The important point before the analysis of the effects can start is that *the stakeholders agree to and understand the scale*.



When you perform the analysis of the effect of the risks you have identified, you must keep your focus on the effect. You must *NEVER let the probability influence the effect!* It can sometimes be tempting to give the effect an extra little turn upwards if we know (or think) that the probability of the risk materializing is high. This will give a twisted picture of the risk exposure and should be avoided.

A simple effect analysis for the risks pertaining to the four top-level architectural areas defined for a product may look like this, using a scale from 1–6 where 6 is worst.

Risk area	Effect
Setup	2
Conveyor	2
Concentration calculation	6
Compound determination	5

**Ex.**

Often it is not enough to have one single score for the effect. Stakeholders see the effect from different perspectives. An end user sees the effect in the light of how a failure will influence his or her daily work. A customer may look at the effect of failures on the overall business goals. A supplier organization may assess the effect in terms of correct efforts for failures or loss of credibility in the market.

These different perspectives can be honored if we use a more complex or composite effect analysis. The score should be the same for all the perspectives, but the descriptions should be tailored to make sense for each of the viewpoints.

A composite effect analysis taking more perspectives into account may look like this:

Risk area	Effect for perspective			Final effect
	User	Customer	Supplier	
Setup	5	3	2	3.3
Conveyor	3	3	5	3.7
Concentration calculation	2	5	2	3
Compound determination	1	5	3	3

**Ex.**

Here all perspectives have the same weight, and the final effect is a simple average of the effect contributions.

If the scale is not sufficiently differentiated the individual perspectives may be assigned independent weights, and the final effect can then be calculated as the weighted average:

$$\text{Final effect} = \sum(\text{effect} * \text{weight}) / \sum(\text{weight})$$

The effect analysis taking more perspectives into account and assigning different weights to the perspectives may look as shown next.



**Ex.**

Risk area	Effect for perspective			Final effect
	User W=2	Customer W=7	Supplier W=1	
Setup	5	3	2	3.3
Conveyor	3	3	5	3.2
Concentration calculation	2	5	2	4.1
Compound determination	1	5	3	3.9

### 3.5.3.5 Probability

The probability is the likelihood of the materialization of a risk.

Also here we first of all need to agree on a scale. On a *quantitative scale* probability can be measured on a scale from 0 to 1 or a scale from 0% to 100%. For most risks it is, however, almost impossible to determine the probability with such a precision.

A *qualitative scale* for probability is usually much more useful, as long as it doesn't get too loose, like "likely," "very likely," "extremely likely."

A qualitative scale could be expressed as in the following table where there is a column for probability intervals, a column describing the probability, and a column for the actual score. Again using a numeric score makes it possible to calculate the risk exposure even when a qualitative scale is used for the probability.

**Ex.**

Probability	Description	Score
99–82	Highly likely	6
81–	Likely	5
–50	Above 50–50	4
49–	Below 50–50	3
Low	Unlikely	2
–1	Highly unlikely	1

Inspired  
by Paul Gerrard



For effect, you must keep your focus on the probability when you perform the analysis of the probability of the risks you have identified. You must *NEVER let the effect influence the probability!* It can sometimes be tempting to give the probability an extra little turn upwards if we know (or think) that the effect of the risk if it materializes is high. This will give an untrue picture of the risk exposure and should be avoided.

The probability of a risk materializing may be a function of many factors, for example:

- ▶ Complexity of the product or the code
- ▶ Size of the product or the code
- ▶ The producer of the work product(s) or component(s)
- ▶ Whether it is a new product or code or maintenance
- ▶ The previous defect record for the product or area
- ▶ The developers' familiarity with tools and processes

Just like it is explained for the effect above the final probability can be calculated as the weighted average of the probabilities pertaining to the different factors.

$$\text{Final probability} = \sum(\text{probability} * \text{weight}) / \sum(\text{weight})$$

A composite probability analysis may look like this:

Risk area	Probability for factor			Final prob.
	Size W=1	History W=5	Complexity W=2	
Setup	4	2	1	2
Conveyor	5	3	5	3.8
Concentration calculation	3	1	2	1.5
Compound determination	3	1	5	2.2

*The same quantitative scale must be used for all the factors.*

### 3.5.3.6 Risk Level

The risk level is calculated for each of the identified risks as

$$\text{Risk level} = \text{final effect} \times \text{final probability}$$

It is not a difficult task to perform a risk analysis as explained above. A full analysis including identifying about 30 risks and assessing and calculating the effect, probability, and final level can be done in a couple of hours. It is well worth the effort because it gives everybody involved a much clearer picture of why test is necessary and how the test should be planned.

Spreadsheets can be used for easy calculation of the levels and for maintenance of the risk analysis.

Using the above examples for final effect and final probability, the final risk level may look as shown in the following table.

**Ex.**

Risk area	Final effect	Final probability	Final risk level
Setup	3.3	2	<b>6.6</b>
Conveyor	3.3	3.8	<b>12.2</b>
Concentration calculation	4.1	1.5	<b>6.2</b>
Compound determination	3.9	2.2	<b>8.6</b>

It sometimes happens that some *stakeholders are unhappy with the final level*. If a stakeholder has high rates for a particular risk and the risk comes out with a relatively low final risk level, this can “seem unfair.” In such cases the perspectives and the scales will have to be discussed once more.

The point of the perspectives and the scales is that they should satisfy every stakeholder’s viewpoint. If that is not the case they must be adjusted. Most of the time, however, stakeholders recognize that the perspectives and scales are OK and that their viewpoint is fairly overruled by others, different viewpoints.



*The distribution of the final risk level over individual risks is used to plan the test activities.* It can be used to prioritize the test activities and to distribute the available time and other resources according to the relative risk level. A test plan based on a risk analysis is more trusted than a plan based on “gut feeling.”



It is difficult to predict events, and therefore all risk analysis has some built-in uncertainty. *A risk analysis must be repeated at regular intervals as the testing progresses.*

The testing results can be used as input to the continuous risk analysis. If we get more defects than expected in a particular area it means that the probability is higher than we expected, and the area hence has a higher risk level. On the other hand if we get fewer defects than expected the risk level is lower.

### 3.5.4 Risk Mitigation

We use the results of the risk analysis as the basis for the risk mitigation, the last activity of the sequential risk management activities. “To mitigate” means “to make or become milder, less severe or less painful.” That is what we’ll try to do.

Faced with the list of risks and their individual risk level we have to go through each of the risks and decide:

- ▶ What we are going to do
- ▶ How we are going to do it
- ▶ When we are going to do it

### 3.5.4.1 What to Do to Mitigate Risks

In terms of what to do we have the following choices:

- ▶ Do nothing
- ▶ Share the pain
- ▶ Plan contingency action(s)
- ▶ Take preventive action



We can choose to do nothing if the benefit of waiting to see how things develop is greater than the cost of doing something.

You would not buy a safe for € 1,000 to protect your jewels if they were only worth € 500 (including the sentimental value). If the jewels were stolen you could buy new ones and still have money left.



Sharing the pain is outside the scope of testing, but it is a possibility for the project management or higher management to negotiate sharing the pain of the effect of a materializing risk with other parties. This other party could be an insurance company or it could be a supplier or even the customer.

Planning contingency action is a natural part of most risk mitigation. The contingency action is what we are going to do to mitigate the effect of a risk once it actually has materialized. For other risk types than product risks and other processes than testing the response to the risk analysis may be production of contingency plans. But it is not something that is applicable in the test planning for mitigating product risks.

Extra courses are planned if it turns out that the system is more difficult to learn than expected.



Testing is one of a number of possible preventive actions. The aim is to mitigate the risks. Testing can be used to mitigate the risk exposure by lowering or eliminating the probability of the risk.

Product risks are associated with presence of defects. The effect is associated with the effect if a defect causes a failure of the product in use. The probability is associated with the probability of undetected defects still being present in the product when it is released.

Testing aims at identifying defects by making the product fail—before it reaches the customer. Defects found in testing can be corrected—before the system reaches the customer. Hence testing and defect correction reduces the risk level by reducing the probability.

### 3.5.4.2 How to Mitigate Risks by Testing

When we have decided to do something to mitigate a risk, we must find out what to do. The nature of the risk can be used to determine the phases and types of testing to perform to mitigate the risk and the level of formality applied. The decisions must be documented in the applicable test strategy or test plan.

Certain test phases are especially applicable for certain types of risks. We need to look at the risk source and determine the phases and activities that are most likely to unveil defects.

Some examples are given next.



Risk source	Recommended test phases
High risk of defects in algorithms	<ul style="list-style-type: none"> <li>▶ Review of detailed design</li> <li>▶ Review of code</li> <li>▶ Component testing</li> <li>▶ Functional system test</li> </ul>
Risk of problems in the user interface	<ul style="list-style-type: none"> <li>▶ Usability evaluation of prototype (requirements review)</li> <li>▶ Usability test (nonfunctional system test)</li> </ul>
Risk of performance problems	<ul style="list-style-type: none"> <li>▶ Performance test (nonfunctional system test)</li> </ul>
Risk concerning external interface	<ul style="list-style-type: none"> <li>▶ Review of design</li> <li>▶ Review of code</li> <li>▶ System integration test</li> </ul>

The formality of the test can also be determined from the risk exposure. The rule is simple:

The higher the risk level => The higher the formality



The formality can change from level to level and it can change over the product. Some areas can have more formal testing than others, even within the same test level.

At any level, for example, system testing, we can have the different levels for formality as shown in the following table.

**System test**

Risk level	Recommended test phases
High	<ul style="list-style-type: none"> <li>▷ Specific test case design techniques to be used</li> <li>▷ Strict test completion criteria</li> <li>▷ Strict regression test procedures</li> </ul>
Low	<ul style="list-style-type: none"> <li>▷ Free choice of test case design techniques</li> <li>▷ Less strict test completion criteria</li> <li>▷ Less strict regression test procedures</li> </ul>

**Ex.****3.5.4.3 When to Mitigate Risks by Testing**

We can use the results of the risk analysis to prioritize the test activities that we have identified for the risks and to distribute the test time (and possibly other resources).

In the prioritization we are going to determine the order in which to attack the risks. Even if we are not going to perform all the testing activities identified for the risks in strictly sequential order, it is a help in the planning to have them prioritized.

The priority can follow the final risk exposure. This means that the final exposure can be as the sorting criteria. This takes every perspective of the risks into consideration in one attempt.

With the example from above the priority of the risk areas can then be as shown in the following, where 1 is the highest.

Risk area	Final risk level	Priority
Setup	<b>6.6</b>	3
Conveyor	<b>12.2</b>	1
Concentration calculation	<b>6.2</b>	4
Compound determination	<b>8.6</b>	2

**Ex.**

The stakeholders could also choose to let the prioritization be guided by either the final effect or the final probability, or they may even agree to use one particular perspective, for example, the probability related to the complexity, to prioritize from.

In order to calculate the distribution of the time to spend on the testing we need to calculate the sum of the final exposure.

**Ex.**

Risk area	Final level
Setup	6.6
Conveyor	12.2
Concentration calculation	6.2
Compound determination	8.6
<b>Total</b>	<b>33.6</b>

The next step is to calculate the distribution of the final levels over the risk areas. This could look as shown here, where the percentages have been rounded to the nearest whole number.

**Ex.**

Risk area	Final level	% distribution
Setup	6.6	20 %
Conveyor	12.2	36 %
Concentration calculation	6.2	10 %
Compound determination	8.6	26 %
<b>Total</b>	<b>33.6</b>	<b>100 %</b>

With a table like this we have a strong planning tool. No matter which resource we have at our disposal we can distribute it on the risk areas and hence ensure that each area is indeed tested, but neither more nor less than it deserves.

If the project manager for example allocates 400 hours for the complete test of our example system, we can distribute this time over the areas as shown here:

**Ex.**

Risk area	% distribution	Hours for testing
Setup	20 %	80
Conveyor	36 %	144
Concentration calculation	18 %	72
Compound determination	26 %	104
<b>Total</b>	<b>100</b>	<b>400</b>

The list of prioritized risks with their allocated resources and identified testing phases and activities allows us to produce a substantiated plan and schedule for the test.

The list also allows us to immediately assess the results of a proposed change in resource allocation. If the resource we have distributed is cut, we will have to find out how to make do with what is left.

Usually we are operating with time; having a number of hours allocated for the testing and consequently a number of hours may be cut. If time is cut we must ask management what to do with our distribution of time on the risks. We can't leave our plan and schedule untouched; the cut must have an effect. What we can do is, in principle:

- ▶ Reduce testing time proportionally to the cut
- ▶ Take risk areas out of the testing completely



The best thing to do is to reduce the time proportionally. This ensures that all areas are still tested, that is, we will get some information relating to all the risks. We can combine the two approaches but we should be very careful if we take areas out completely.

If some testing has already been performed, a renewed risk analysis is necessary before we can act on any cuts. In this case we must distribute the remaining resources over the remaining risks according the new final exposure and prioritize as we did before or by a new, more relevant criterion.

## Questions

1. How does testing provide business value?
2. What is the purpose of testing?
3. What is product reliability?
4. On what does the cost of defect correction depend?
5. Where do most of the defects originate from?
6. What does good testing provide?
7. What four types of test management documents are defined?
8. What is the purpose of the test policy?
9. What should the test policy include?
10. What could a quality target for example be?
11. What is the purpose of the test strategy?
12. What are the approaches to the testing?
13. What are the strategy topics to be considered?
14. Why may standards be useful?
15. What should a strategy include in relation to risks?
16. Why are completion criteria important?
17. What is the idea in degree of independence?
18. What may be reused in testing?
19. Why does the strategy need to be specific about tools?
20. Why should we measure during testing?
21. To which support process does the incident management belong?
22. What is a master plan for?

23. What is a level test plan?
24. What are the 14 topics that should be covered in a test plan according to IEEE 829?
25. What should the introduction contain?
26. What is the relationship between the test item and features to be tested?
27. Why is it important to describe what is not tested?
28. What should the approach description consider?
29. For what are pass/fail criteria used?
30. What may cause a pause in the test execution?
31. What are the typical test deliverables?
32. What is the important thing concerning testing tasks?
33. How can you illustrate who carries which responsibility?
34. In which part of the test plan do the testing tasks, the estimates, and the people come together?
35. Who should approve the test plan?
36. What does it mean that a (test) plan must be SMART?
37. What is *The New Yorker* way of planning?
38. Who may be interested in test reports?
39. What topics should be covered in a test summary report according to IEEE 829?
40. What is a comprehensiveness assessment?
41. Who must decide if a test object should be released?
42. What is estimation?
43. What is not estimation?
44. How is test estimation different from ordinary project estimation?
45. What are the six steps in the estimation process?
46. What are the three estimation technique types?
47. On what is the analogy technique based?
48. What are the steps in the Delphi technique?
49. What three estimates do we need for the successive calculation estimation technique?
50. What is the estimation based on in the successive calculation estimation technique?
51. What are the five things you count for function point calculation?
52. What is the test estimation technique based on function points?
53. What is the difference between all the other techniques and the percentage distribution technique?
54. What must be taken into account when defining the finish date for a task?
55. When can we stop estimating?

56. What must guide our follow-up activities?
57. What is the virtue in planning?
58. On what should we base our follow-up?
59. How can we present measurements?
60. What is the ink-factor?
61. What is the principle in S-curves?
62. What is a check sheet?
63. What is risk-based reporting?
64. Why should we use statistical reporting for process performance?
65. What must we do to stay in control?
66. What can we do?
67. What is the golden rule of testing?
68. What is a risk?
69. What is risk exposure?
70. What is a process risk?
71. What is a project risk?
72. What could a project risk be?
73. What is a product risk?
74. How does testing relate to risks?
75. What are the activities in risk management?
76. Which stakeholders could be involved in the work with risks?
77. What is a risk template?
78. What could a product risk be?
79. Which risk identification techniques could we use?
80. For what is a risk checklist used?
81. What is the main rule for brainstorms?
82. How must an expert interview be prepared?
83. What is the output from a risk interview?
84. What is risk analysis?
85. How may the viewpoints of risk differ for different roles?
86. What kinds of scales can be used for risk analysis?
87. What must not be done when assessing effect?
88. What is a composite effect analysis?
89. How do you calculate the final effect?
90. What must not be done when assessing probability?
91. Which factors could be used in composite probability assessment?
92. For what is the result of a risk analysis used?
93. How many times should a risk analysis be performed, and why?
94. What is mitigation?
95. What are the actions that can be taken as a result of a risk analysis?
96. Which action is most appropriate for product risks?

97. Which test activities and test levels could be used to mitigate different risks?
98. How are the formality of testing and risk exposures related?
99. How can you prioritize the testing for risks?
100. How can you use risk exposure to distribute testing resources?
101. What can you do if testing time is cut?

## CHAPTER

# 4

## Test Techniques

Test case design techniques are the heart of testing. There are many advantages of using techniques to design test cases. They support systematic and meticulous work and make the testing specification effective and efficient; they are also extremely good for finding possible faults. Techniques are the synthesis of “best practice”—not necessarily scientifically based, but based on many testers’ experiences.

Other advantages are that the design of the test cases may be repeated by others, and that it is possible to explain how test cases have been designed using techniques. This makes the test cases much more trustworthy than test cases just “picked out of the air.” The test case design techniques are based on models of the system, typically in the form of requirements or design. We can therefore calculate the coverage we are obtaining for the various test design techniques.

Coverage is one of the most important ways to express what is required from our testing activities. It is worth noticing that coverage is always expressed in terms related to a specific test design technique. Having achieved a high coverage using one technique only says something about the testing with that technique, not the complete testing possible for a given object.

Test case design techniques have a few pitfalls that we need to be aware of. Even if we could obtain 100% coverage of what we set out to cover (be it requirements, or statements, or paths), faults could remain after testing simply because the code does not properly reflect what the users and customers want. Validation of the requirements before we start the dynamic testing can mitigate this risk.

There is also a pitfall in relation to value sensitivity. Even if we use an input value that gives us the coverage we want it may be a value for which incidental correctness applies. An example of this is the fact that:

$2 + 2$  equals  $2 * 2$ ; but  $3 + 3$  does not equal  $3 * 3!$

## Contents

- 4.1 Specification-Based Techniques
- 4.2 Structure-Based Techniques
- 4.3 Defect-Based Techniques
- 4.4 Experience-Based Testing Techniques
- 4.5 Static Analysis
- 4.6 Dynamic Analysis
- 4.7 Choosing Testing Techniques

## 4.1 Specification-Based Techniques

The specification-based case design techniques are used to design test cases based on an analysis of the description of the product without reference to its internal workings. These techniques are also known as black-box tests.

These techniques focus on the functionality. They are dependent on descriptions of the expectations towards the product or system. These should be in the form of requirements specifications, but may also be in the form of, for example, user manuals and/or process descriptions. If we are lucky we get the requirements expressed in ways corresponding directly to these techniques; if not we'll have to help analysts do that during requirements documentation or do it ourselves during test design.

These test case design techniques can be used in all stages and levels of testing. The techniques can be used as a starting point in low-level tests, component testing and integration testing, where test cases can be designed based on the design and/or the requirements. These test cases can be supplied with structural or white-box tests to obtain adequate coverage.

The techniques are also very useful in high-level tests like acceptance testing and system testing, where the test cases are designed from the requirements.

The specification-based techniques have associated coverage measures, and the application of these techniques refines the coverage from requirements coverage to specific coverage items for the techniques.

The functional test case design techniques covered here are:



- ▶ Equivalence partitioning and boundary value analysis
- ▶ (Domain analysis—not part of the ISTQB syllabus)
- ▶ Decision tables
- ▶ Cause-effect graph
- ▶ State transition testing
- ▶ Classification tree method
- ▶ Pairwise testing
- ▶ Use case testing
- ▶ (Syntax testing—not part of the ISTQB syllabus)

### 4.1.1 Equivalence Partitioning and Boundary Value Analysis

Designing test cases is about finding the input to cover something we want to test. If we consider the number of different inputs that we can give to a product we can have anything from very few to a huge amount of possibilities.

A product may have only one button and it can be either on or off = 2 possibilities.

A field must be filled in with the name of a valid postal district = thousands of possibilities.

It can be very difficult to figure out which input to choose for our test cases. The equivalence partitioning test technique can help us handle situations with many input possibilities.

#### 4.1.1.1 Equivalence Partitioning

The basic idea is that we partition the input or output domain into equivalence classes.

A class is a portion of the domain. The domain is said to be partitioned into classes if all members of the domain belong to exactly one class—no member belongs to more than one class and no member falls outside the classes.



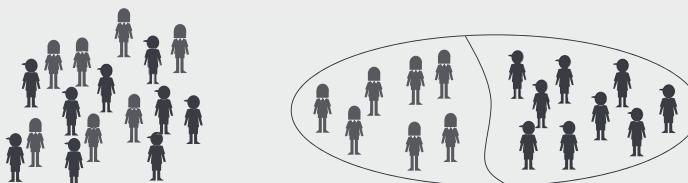
The term equivalence refers to the assumption that all the members in a class behave in the same way. In this context the assumption is based on the requirements or other specifications of the product's expected behavior.



The reason for the equivalence partitioning is that all members in an equivalence class will either fail or pass the same test. One member represents all! If we select one member of a class and use that for our test case, we can assume that we have tested all the members.

Choosing test cases based on equivalence partitioning insures representative test cases.

If we take a class of pupils and the requirement says that all the girls should have an e-mail every Thursdays reminding them to bring their swimsuits, we can partition the class into a partition of girls and a partition of boys and use one representative from each class in our test cases.



When we partition a domain into equivalence classes, we will usually get both valid and invalid classes. The invalid classes contain the members that the product should reject, or in other words members for which the product's behavior is unspecified. Test cases should be designed for both the valid and the invalid classes, though sometimes it is not possible to execute test cases based on the invalid equivalence classes.

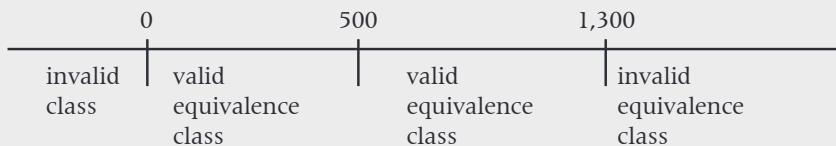
The most common types of equivalence class partitions are intervals and sets of possibilities (unordered list or ordered lists).

**Ex.**

Intervals may be illustrated by a requirement stating:

Income in €	Tax percentage
Up to and including 500	0
More than 500, but less than 1,300	30
1,300 or more, but less than 5,000	40

If this is all we know, we have:



Another invalid equivalence class may be inputs containing letters.

**Ex.**

To illustrate a set of possibilities we may use the unordered list of hair colors: (blond, brown, black, red, gray). Perhaps the product can suggest an appropriate dye for these colors of hair, but none other. The valid equivalence class is the list of values; all other values belong to the invalid class, assuming we don't know how the product is going to react, if we enter one such value.



It is possible to measure the coverage of equivalence partitions. The equivalence partition coverage is measured as the percentage of equivalence partitions that have been exercised by a test.

To exercise an equivalence class we need to pick one value in the equivalence class and make a test case for this. It is quite usual to pick a value near the middle of the equivalence class, if possible, but any value will do.

**Ex.**

Test cases for the tax percentage could be based on the input values: -5; 234; 810; and 2,207.

For the hair colors we could choose: black and green, as a valid and an invalid input value, respectively.



#### 4.1.1.2 Boundary Value Analysis

A boundary value is the value on a boundary of an equivalence class. Boundary value analysis is hence strongly related to equivalence class partitioning. Equivalence classes of intervals have boundaries, but those of lists do not. Boundary value analysis is the process of identifying the boundary values.

The boundary values require extra attention because defects are often found on or immediately around these. Choosing test cases based on boundary value analysis insures that the test cases are effective.

For interval classes with precise boundaries, it is not difficult to identify the boundary values.

The interval given as:  $0 \leq \text{income} \leq 500$ , is one equivalence class with the two boundaries 0 and 500.



If a class has an imprecise boundary ( $>$  or  $<$ ) the boundary value is one increment inside the imprecise boundary.

If the above interval had been specified as:  $0 \leq \text{income} < 500$ , and the smallest increment is given as 1, we would have an equivalence class with the two boundaries 0 and 499.



The smallest increment should always be specified; otherwise we must ask or guess based on common or otherwise available information.

It is often not specified what the increment is when we are dealing with money. A reasonable guess is that if we operate in euros (€), then the smallest increment is € 0.01 or 1 cent. But beware! Tax people usually use € 1 as the smallest increment.



Sometimes we'll experience equivalence classes with open boundaries—classes where a boundary is not specified. This makes it difficult to identify a boundary value to test. In these cases we must first of all try to get the specification changed. If that is not possible we can look for information in other requirements, look for indirect or hidden boundaries, or omit the testing of the nonexistent boundary value.



Open boundaries may be seen in connection with people's incomes. If the above interval had to do with incomes we may argue that the lowest income is € 0. But what is the real upper boundary? There is no obvious upper boundary for people's income.



Because we can have both valid and invalid equivalence classes we can also have both valid and invalid boundary values, and both should be tested. Sometimes, however, the invalid ones can not be tested.

When we select boundary values for testing, we must select the boundary value and at least one value one unit inside the boundary in the equivalence class. This means that for each boundary we test two values.



In traditional testing it was also recommended to choose a value one unit



outside the equivalence class, hence testing three values for each boundary. Such a value is in fact a value on the border of the adjacent equivalence class, and some duplication could occur. But we can still choose to select three values; the choice between two or three values should be governed by a risk evaluation.



It is possible to measure the coverage of boundary values. The boundary value coverage is measured as the percentage of boundary values that have been exercised by a test.

### ***Equivalence Partitioning and Boundary Value Analysis Test Design Template***

The design of the test conditions based on equivalence class partitioning and boundary value analysis can be captured in a table such as the following one.

<b>Test design item number:</b>		<b>Traces:</b>	
<b>Based on: Input/Output</b>		<b>Assumptions:</b>	
<b>Type</b>	<b>Description</b>	<b>Tag</b>	<b>BT</b>

Table designed  
by Carsten  
Jørgensen

The fields in the table are:



*Test design item number:* Unique identifier of the test design item

*Traces:* References to the requirement(s) or other descriptions covered by this test design

*Based on: Input/Output:* Indication of which type of domain the design is based on

*Assumptions:* Here any assumption must be documented.

For each test condition we have the following fields:

*Type:* Must be one of

- ▶ VC—Valid class
- ▶ IC—Invalid class
- ▶ VB—Valid boundary value
- ▶ IB—Invalid boundary value



Remember that the invalid values should be rejected by the system.

*Description:* The specification of the test condition

*Tag:* Unique identification of the test condition

*BT = Belongs to:* Indicates the class a boundary value belongs to. This can be used to cross-check the boundary values.

***Equivalence Partitioning and Boundary Value Analysis Test Design Examples***

In this example we shall find test conditions and test cases for the testing of this user requirement.

[UR 631] The system shall allow shipments for which the price is less than or equal to € 100.

The first thing we'll do is fill in the header of the design table.

**Ex.**

<b>Test design item number: 11</b>	<b>Traces: [UR 631]</b>
<b>Based on:</b> Input	<b>Assumptions:</b> The price cannot be negative The smallest increment is 1 cent

The next thing is identifying the valid class(es).

Type	Description	Tag	BT
VC	0 <= shipment price <= 100		

We then consider if there are any invalid classes. If we only have the single requirement given above, we can identify two obvious and two special invalid equivalence classes. The new rows are indicated in bold.

Type	Description	Tag	BT
<b>IC</b>	<b>shipment price &lt; 0</b>		
VC	0 <= shipment price <= 100		
<b>IC</b>	<b>shipment price &gt; 100</b>		
<b>IC</b>	<b>shipment price is empty</b>		
<b>IC</b>	<b>shipment price contains characters</b>		

Our boundary value analysis gives us two boundary values.

Type	Description	Tag	BT
IC	shipment price < 0		
IB	<b>shipment price = -0.01</b>		
VB	<b>shipment price = 0.00</b>		
VC	0 <= shipment price <= 100		
VB	<b>shipment price = 100.00</b>		
IC	shipment price > 100		
IB	<b>shipment price = 100.01</b>		
IC	shipment price is empty		
IC	shipment price contains characters		

This concludes the equivalence class partitioning and boundary value analysis for the first requirement. We will complete the table by adding tags and indicating to which classes the boundary values belong.

Test design item number: 11		Traces: [UR 631]	
Based on: Input		Assumptions: The price cannot be negative The smallest increment is 1 cent	
Type	Description	Tag	BT
IC	shipment price < 0	11-1	
IB	shipment price = -0.01	11-2	11-1
VB	shipment price = 0.00	11-3	11-4
VC	0 <= shipment price <= 100	11-4	
VB	shipment price = 100.00	11-5	11-4
IC	shipment price > 100	11-6	
IB	shipment price = 100.01	11-7	11-6
IC	shipment price is empty	11-8	
IC	shipment price contains characters	11-9	

We can now make low-level test cases. If we want 100% equivalence partition coverage and two value boundary value coverage for the requirement, assuming that invalid values are rejected, we get the following test cases:

Tag	Test case	Input Price=	Expected output
11-1	TC1-1	-25.00	rejection
11-2	TC1-2	0.02	rejection
11-2	TC1-3	-0.01	rejection
11-3	TC1-4	0.00	OK
11-3	TC1-5	0.01	OK
11-4	TC1-6	47.00	OK
11-5	TC1-7	99.99	OK
11-5	TC1-8	100.00	OK
11-7	TC1-9	100.01	rejection
11-7	TC1-10	100.02	rejection
11-6	TC1-11	114.00	rejection
11-8	TC1-12	" "	rejection
11-9	TC1-13	"abcd.nn"	rejection

We could choose to omit some of the test cases, especially since we actually get five different test cases covering the same equivalence class.

In the next example we will test the following requirement:

**[UR 627]** The system shall allow the packing type to be specified as either "Box" or "Wrapping paper."

The test design table looks like this after the analysis.

**Ex.**

Test design item number: 15		Traces: [UR 627]	
Based on: Input		Assumptions:	
Type	Description	Tag	BT
VC	"Box" "Wrapping paper"	15-1	
IC	All other texts	15-2	

This type of equivalence class does not have boundaries.

The test cases could be:

Tag	Test case	Input packing type	Expected output
15-1	TC3-1	"Box"	OK
15-2	TC3-2	"Paper"	rejection

It is only necessary to test one of the valid packing types, because it is a member of an equivalence class.

## ***Equivalence Partitioning and Boundary***

### ***Value Analysis Hints***



The equivalence partitioning and boundary value analysis of requirements makes it possible for us to:

- ▶ Reduce the number of test cases, because we can argue that one value from each equivalence partition is enough.
- ▶ Find more faults because we concentrate our focus on the boundaries where the density of defects, according to all experience, is highest.

Sometimes the input or output domains we are testing do not have one-dimensional boundaries as assumed in the equivalence class partitioning and boundary value analysis as described here.

If it is not possible or feasible to partition our input or output domain in one dimension we have to use the technique called domain analysis instead.



### **4.1.2 Domain Analysis**

*Note: This technique is not part of the ISTQB syllabus, but included here because I find it interesting and occasionally useful.*

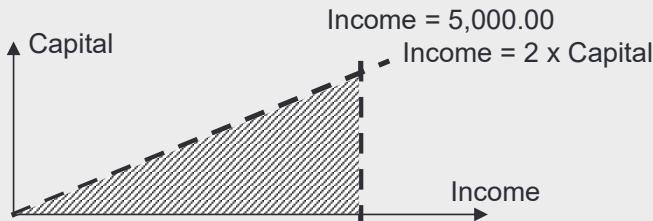
In equivalence partitioning of intervals where the boundaries are given by simple numbers, we have one-dimensional partitions. The domain analysis test case design technique is used when our input partitions are multi-dimensional, that is when a border for an equivalence partition depends on combinations of aspects or variables. If two variables are involved we have a two-dimensional domain; if three are involved, we have a three-dimensional domain, and so on.

Multidimensional partitions are called domains—hence the name of the technique, even though the principles are the same as in equivalence partitioning and boundary value analysis.

It is difficult for people to picture more than three dimensions, but in theory there is no limit to the number of dimensions we may have to handle in domain analysis. We will use two-dimensional domains in the section; the principles are the same for any number of dimensions.

For equivalence partitioning we had the example of intervals of income groups, where  $0.00 \leq \text{income} < 5,000.00$  is tax-free. This is a one-dimensional domain. If people's capital counts in the calculation as well, so that income is only tax-free if it is also less than twice the capital held by the person in question, we have a two-dimensional domain.

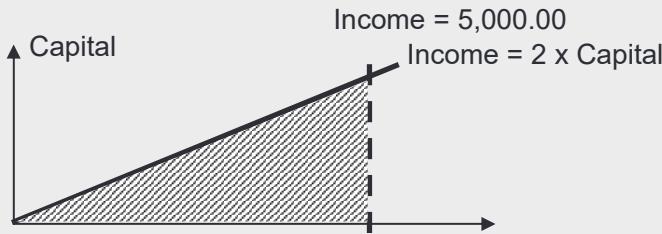
The two-dimensional domain for tax-free income is shown as the striped area (assuming that the capital and the income are both  $\geq 0.00$ ):


**Ex.**

Borders may be either open or closed. A border is *open*, if a value on the border does not belong to the domain we are looking at. This is the case in the example where both the borders are open ( $\text{income} < 5,000.00$  and  $\text{income} < 2 \times \text{capital}$ ).

A border is *closed* if a value on the border belongs to the domain we are looking at.

If we change the border for tax-free income to become:  $\text{income} \leq 2 \times \text{capital}$ , we have a closed border. In this case our domain will look like this:


**Ex.**

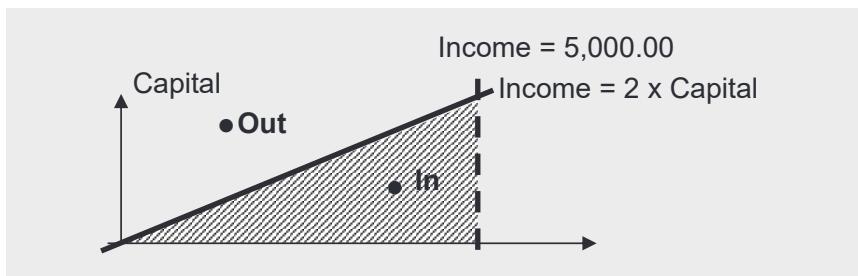
In equivalence partitioning we say that a value is in a particular equivalence class—in a way we do the same for domain analysis, though here we operate with points relative to the borders:

- ▶ A point is an **In** point in the domain we are considering, if it is inside and not on the border
- ▶ A point is an **Out** point to the domain we are considering, if it is outside and not on the border (it is then in another domain)



An In point and an Out point relative to the border income  $\leq 2 \times \text{capital}$  are illustrated here.

**Ex.**



In the boundary value analysis related to equivalence partitioning described above, we operate with the boundary values on the boundary and one unit inside. In domain analysis we operate with On and Off points relative to each border.

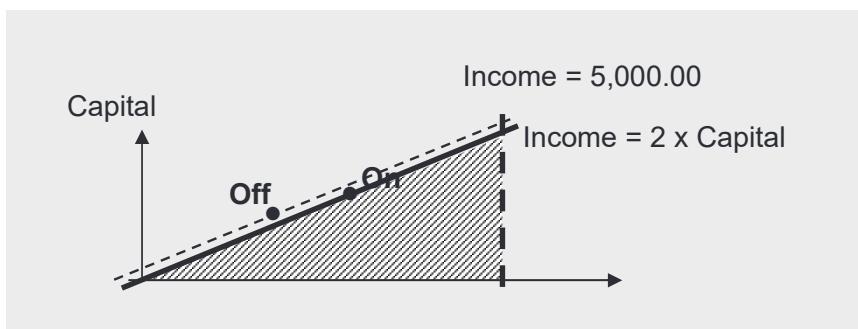
We have:



- A point is an On point if it is on the border between partitions
- A point is an Off point if it is "slightly" off the border

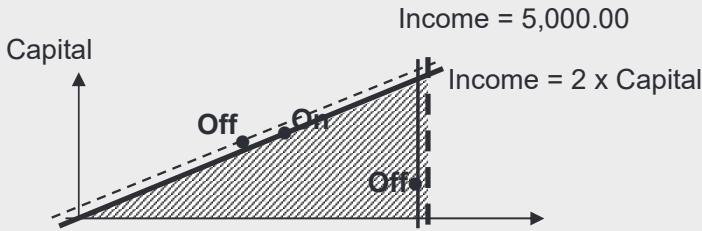
If the border of the domain we are looking at is closed, the Off point will be outside the domain. The "slightly" may be one unit relative to what measure we are using, so that the Off point lies on the border of the adjacent domain. This works for all practical purposes, as long as we are not working with a floating point where "one unit" is undeterminable; in this case the "slightly" will have to be far enough away from the border to ensure that the Off point is inside the adjacent domain.

**Ex.**



If the border of the domain we are looking at is open, the Off point will be "slightly" (or perhaps one unit) inside the domain, namely, outside (or on) the closed border of the adjacent domain.

In our case with the income < 5,000.00 an On point and an Off point may look like this:



### ***Domain Analysis Strategy***

The number of test cases we can design based on a domain analysis depends on the test strategy we decide to follow. A strategy can be described as:

N-On \* N-Off

where N-On is the number of On points we want to test for each border and N-Off correspondingly is the number of Off points we want to test for each border for the domains we have identified.



If we choose a  $1 * 1$  strategy we therefore set out to test one On point and one Off point for each border of the domains we have identified. This is what is illustrated above for the one domain we are looking at (not taking the capital  $\geq 0$  border into account). If our strategy is  $2 * 1$ , we set out to test two On points and one Off point for each of the borders for all the domains we have identified. In this case the On points will be the points where the borders cross each other, that is the extremes of the borders.

In a  $1 * 1$  strategy we will get two test cases for each border. If we are testing adjacent domains, we will get equivalent test cases because an Off point in one domain is an In point in the adjacent domain. These test cases will have identical expected outcomes if identical values are chosen for the low-level test cases. These duplicates need not be repeated in the test procedures for execution.



### ***Domain Analysis Coverage***

It is possible to measure the coverage for domain analysis. The coverage elements for the identified domains are the In points and the Out points. The coverage is measured as the percentage of In points and Out points that have been exercised by a test. Do not count an In point in one partition being an Out point in another partition to be tested twice.



The coverage elements for the borders are the On points and Off points. The coverage is measured as the percentage of On points and Off points that have been exercised by a test relative to what the strategy determines as the number of points to test. Again do not count duplicate points twice.



Table designed  
by Carsten  
Jørgensen



### **Domain Analysis Test Design Template**

The design of the test conditions based on domain analysis and with the aim of getting On and Off point coverage can be captured in a table like this one.

Tag						
Border 1 condition	ON	OFF				
Border 2 condition			ON	OFF		
Border n condition					ON	OFF

The table is for one domain; and it must be expanded both in the length and width to accommodate all the borders our domain may have.

The rule is: divide and conquer.

For each of the borders involved we should:

- ▶ Test an On point
- ▶ Test an Off point

If we want In point and Out point coverage as well, we must include this explicitly in the table.

When we start to make low-level test cases we add a row for each variable to select values for. In a two-dimensional domain we will have to select values for two variables.

Tag						
Border 1 condition	ON	OFF				
Border 2 condition			ON	OFF		
Border n condition					ON	OFF
<hr/>						
Variable X						
Variable Y						

For each column we select a value that satisfies what we want. In the first column of values we must select a value for X and a value for Y that gives us a point On border 1. We should aim at getting In points for the other borders in the column, though that is not always possible. For each of the borders we should note which kind of point we get for the selected set of values.

This selection of values can be quite difficult, especially if we have high-dimensional domains. Making the table in a spreadsheet helps a lot, and other tools are also available to help.

### *Domain Analysis Test Design Example*

In this example we shall test this user requirement:

**[UR 637]** The system shall allow posting of envelopes where the longest side (l) is longer than or equal to 12 centimeters, but not longer than 75 centimeters. The smallest side (w) must be longer than or equal to 1 centimeter. The length must be twice the width and must be greater than or equal to 10 centimeters. Measures are always rounded up to the nearest centimeter. All odd envelopes are to be handled by courier.

**Ex.**

We can rewrite this requirement to read:

length  $\geq 12$

length  $< 75$

width  $\geq 1$

length - 2  $\times$  width  $\geq 10$

This can be entered in our template:

Tag	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
Length $\geq 12$	ON 12	OFF						
Length $< 75$	12 IN		ON	OFF				
Width $\geq 1$	1 IN				ON	OFF		
1 - 2 $\times$ w $\geq 10$	10 ON						ON	OFF
Length	12							
Width	1							

In the table we have entered values for the first test case, namely length = 12 to get the On point for the first border condition. The simultaneous selecting of width = 1 gives the points indicated for each border in lowercase under the number.

The table fully filled in may look like this:

Tag	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
<b>Length &gt; = 12</b>	ON 12	OFF 11	75 in	74 in	15 in	15 in	30 in	31 in
<b>Length &lt; 75</b>	12 IN	11 in	ON 75	OFF 74	15 in	15 in	30 in	31 in
<b>Width &gt; = 1</b>	1 in	1 in	30 in	30 in	ON 1	OFF 0	10 in	11 in
<b>1 – 2 x w &gt; = 10</b>	10 on	9 out	15 in	14 in	13 in	15 in	ON 10	OFF 9
<b>Length</b>	12	11	75	74	15	15	30	31
<b>Width</b>	1	1	30	30	1	0	10	11

We now need to determine the expected results, and then we have our test cases ready.

#### 4.1.3 Decision Tables

A decision table is a table showing the actions of the system depending on certain combinations of input conditions.

Decision tables are often used to express rules and regulations for embedded systems and administrative systems. They seem to have gone a little out of fashion, and that is a shame. Decision tables are brilliant for overview and also for determining if the requirements are complete.

It is often seen that what could have been expressed in a decision table is attempted to be explained in text. The text may be several paragraphs or even pages long and reformatting of the text into decision tables will often reveal holes in the requirements.

Decision tables are useful to provide an overview of combinations of inputs and the resulting output. The combinations are derived from requirements, which are expressed as something that is either true or false. If we are lucky the requirements are expressed directly in decision tables where appropriate.

Decision tables always have  $2^n$  columns, because there are always  $2^n$  combinations, where  $n$  is the number of input conditions.

The number of rows in decision tables depends on the number of input conditions and the number of dependent actions. There is one row for each condition and one row for each action.

Input condition 1	T	T	F	F
Input condition 2	T	F	T	F
Action 1	T	T	F	F
Action 2	T	T	T	F



The table is read one column at a time. We can, for example, see that if both input conditions are true then both actions will happen (be true).

The *coverage measure* for decision tables is the percentage of the total number of combinations of input tested in a test.

Sometimes it is not possible to obtain 100% combination coverage because it is impossible to execute a test case for a combination.



#### 4.1.3.1 Decision Table Templates

The template to capture decision table test conditions is the template for the decision table itself with a test design header, as shown next.

Test design item number:	Traces:			
Assumptions:	TC1	TC2		TCn
Input condition 1				
Input condition n				
Action 1				
Action n				

The fields in the table are:

*Test design item number:* Unique identifier of the test design item



*Traces:* References to the requirement(s) or other descriptions covered by this test design

*Assumptions:* Here any assumption must be documented

The table must have a row for each input and each action, and  $2^n$  columns, where  $n$  is the number of input conditions. The cells are filled in with either True or False to indicate if the input conditions, respectively the actions are true or false.

The easiest way to fill out a decision table is to fill in the input condition rows first. For the first input condition half of the cells are filled with True and the second half are filled with False. In the next row half of the cells under the Ts are filled with True and the other half with False, and likewise for the Fs. Keep on like this until the Ts and Fs alternate for each cell in the last input condition row.

The values for the resulting actions must be extracted from the requirements!



#### 4.1.3.2 Decision Table Example

**Ex.**

In this example we are going to test the following requirements.

[76] The system shall only calculate discounts for members.

[77] The system shall calculate a discount of 5% if the value of the purchase is less than or equal to € 100. Otherwise the discount is 10%.

[78] The system shall write the discount % on the invoice.

[79] The system must write in the invoices to nonmembers that membership gives a discount.

Test design item number: 82	Traces: Req. [76]–[79]			
<b>Assumptions:</b> The validity of the input is tested elsewhere				

Note that we are only going to test the calculation and printing on the invoice, not the correct calculation of the discount.

	TC1	TC2	TC3	TC4
Purchaser is member	T	T	F	F
Value <= € 100	T	F	T	F
No discount calculated	F	F	T	T
5% discount calculated	T	F	F	F
10% discount calculated	F	T	F	F
Member message on invoice	F	F	T	T
Discount % on invoice	T	T	F	F

The test cases to execute can now be created from these test conditions and written directly into a test procedure.

Test procedure: 11			
Purpose: This test procedure tests the calculation of discount for members.			
Traces: Req. [76]–[79]			
Tag	TC	Input	Expected output
TC1	1	Choose a member and create a purchase with a value less than € 100	A discount of 5% is calculated and this is written on the invoice.
TC2	2	Choose a member and create a purchase with a value of more than € 100	A discount of 10% is calculated and this is written on the invoice.
TC3	3	Choose a nonmember and create a purchase with a value less than € 100	No discount is calculated and the “membership gives discount” statement is written on the invoice.

### 4.1.3.3 Collapsed Decision Tables

Sometimes it seems evident in a decision table that some conditions are without effect because one decision is decisive. For example if one condition is False an action seems to be False no matter what the values of all the other conditions are.

This could lead us to collapse the decision table, that is reduce the number of combinations by only taking one of those where the rest will give the same result. This technique is related to the condition determination testing discussed below in Section 4.2.6.

The decision as to whether to collapse a decision table or not should be based on a risk analysis.



### 4.1.4 Cause-Effect Graph

A cause-effect graph is a graphical way of showing inputs or stimuli (causes) with their associated outputs (effects). The graph is a result of an analysis of requirements. Test cases can be designed from the cause-effect graph.

The technique is a semiformal way of expressing certain requirements, namely requirements that are based on Boolean expressions.

The cause-effect graphing technique is used to design test cases for functions that depend on a combination of more input items.

In principle any functional requirement can be expressed as:

$$f(\text{old state, input}) \rightarrow (\text{new state, output})$$



This means that a specific treatment ( $f = \text{a function}$ ) for a given input transforms an old state of the system to a new state and produces an output.

We can also express this in a more practical way as:

$$f(\text{ops1, ops2,..., i1, i2..i}) \rightarrow (\text{ns1, ns2,..o1, o2..})$$


where the old state is split into a number of old partial states, and the input is split into a number of input items. The same is done for the new state and the output.

The causes in the graphs are characteristics of input items or old partial states. The effects in the graphs are characteristics of output items or new partial states.

Both causes and effects have to be statements that are either True or False. True indicates that the characteristic is present; False indicates its absence.



The graph shows the connections and relationships between the causes and the effects.

#### 4.1.4.1 Cause-Effect Graph Coverage

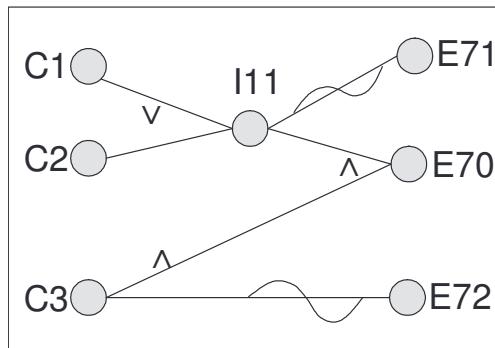
The coverage of the cause-effect graph can be measured as the percentage of all the possible combinations of inputs tested in a test suite.

#### 4.1.4.2 Cause-Effect Graphing Process and Template

A cause-effect graph is constructed in the following way based on an analysis of selected suitable requirements:

- ▶ List and assign an ID to all causes
- ▶ List and assign an ID to all effects
- ▶ For each effect make a Boolean expression so that the effect is expressed in terms of relevant causes
- ▶ Draw the cause-effect graph

An example of a cause-effect graph is shown here.



The graph is composed of some simple building blocks:



Identified cause or effect—Must be labeled with the corresponding ID. It is a good idea to start the IDs of the causes with a C and those of the effects with an E. Intermediate causes may also be defined to make the graph simpler.

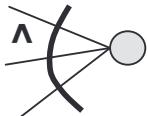


Connection between cause(s) and effect—The connection always goes from the left to the right.

∧ This means that the causes are combined with AND, that is all causes must be True for the effect to be True.

∨ This means that the causes are combined with OR, that is only one cause needs to be True for the effect to be True.

 This is a negation, meaning that a True should be understood as a False, and vice versa.



The arch shows that all the causes (to the left of the connections) must be combined with the Boolean operator; in this case the three causes must be "AND'et."

Test cases may be derived directly from the graph. The graph may also be converted into a decision table, and the test cases derived from the columns in the table.

Sometimes constraints may apply to the causes and these will have to be taken into consideration as well.

#### 4.1.4.3 Cause-Effect Graph Example

In this example we are going to test a Web page, on which it is possible to sign up for a course. The Web page looks like this:

**Ex.**

**Sign up for course**

Name:

Address:

Zip Code:  City:

Course Number:

First we make a complete list of causes with identification. The causes are derived from a textual description of the form (not included here):

- C1. Name field is filled in
- C2. Name contains only letters and spaces
- C3. Address field is filled in
- C4. Zip code is filled in
- C5. City is filled in
- C6. Course number is filled in
- C7. Course number exists in the system

An intermediate Boolean may be introduced here, namely I30 meaning that all fields are filled in. This is expressed as:

$$I30 = \text{and} (C1, C3, C4, C5, C6)$$

The full list of effects with identification is:

E51. Registration of delegate in system

E52. Message shown: All fields should be filled in

E53. Message shown: Only letters and spaces in name

E54. Message shown: Unknown course number

E55. Message shown: You have been registered

We must now express each effect as a Boolean expression based on the causes. They are:

$$E51 = \text{and}(I30, C2, C7)$$

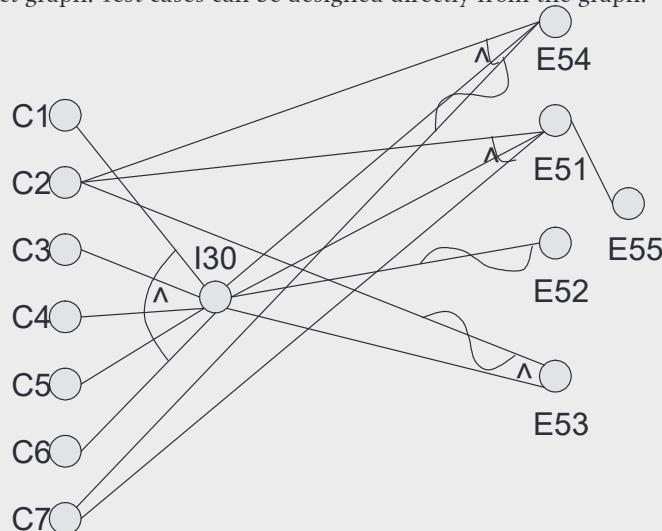
$$E52 = \text{not } I30$$

$$E53 = \text{and}(I30, \text{not } C2)$$

$$E54 = \text{and}(I30, C2, \text{not } C7)$$

$$E55 = E51$$

Drawing the causes and the effects and their relationships gives us the cause-effect graph. Test cases can be designed directly from the graph.



#### 4.1.4.4 Cause-Effect Graph Hints

The cause-effect graph test case design technique is very suitable for people with a graphical mind.

For others it may be a help in the analysis phase and the basis for the construction of a decision table from which test cases can be described as discussed above.

All the effects can be filled into the decision table by looking at the cause-effect graph or even from the Boolean expressions directly. Fill in all combinations of True and False for the causes, and then fill in the impact each combination has on the effects.

Cause-effect graphs frequently become very large—and therefore difficult to work with. To avoid this divide the specification into workable pieces of isolated functionality.

To mitigate the size problem we can select different ways to reduce the decision table, such as removal of impossible combinations. We can also reduce the combinations by only keeping those that independently affect the outcome, like in condition determination testing, discussed in Section 4.2.6.



#### 4.1.5 State Transition Testing

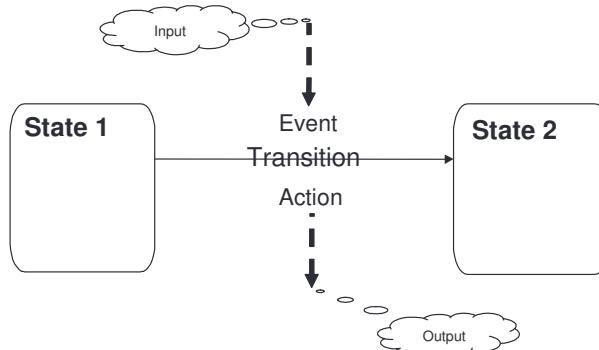
State transition testing is based on a state machine model of the test object. State machine modeling is a design technique, most often used for embedded software, but also applicable for user interface design. If we are lucky we get the state machine model as part of the specification we are going to test against. Otherwise we will have to extract it from the requirements.

Most products and software systems can be modeled as a state machine. The idea is that the system can be in a number of well-defined states. A state is the collection of all features of the system at a given point in time, including all visible data, all stored data, and any current form and field.

The transition from one state to another is initiated by an event. The system just sits there doing nothing until an event happens. An event will cause an action and the object will change into another state (or stay in the same state).

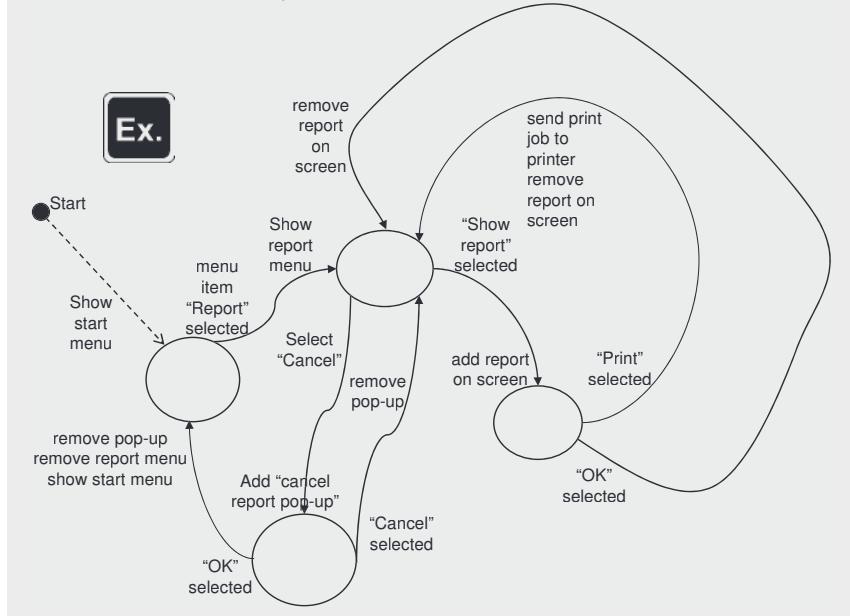
A transition = start state + event + action + end state

The principle in a state machine is illustrated next.



State machines can be depicted in many ways. The figure below shows a state machine presentation of a report printing menu, where the states are depicted as circles and the events and actions are written next to the transitions depicted as arrows.

It is a good idea to leave the states blank and just give them an identification, typically in the form of a number. The events, actions, and transitions should also be identified.



Note that the state machine has a start state. This could be a transition from another state machine describing another part of the full system.

#### 4.1.5.1 State Transition Testing Coverage

Transitions can be performed in sequences. The smallest “sequence” is one transition at a time. The second smallest sequence is a sequence of two transitions in a row. Sequences can be of any length.

The coverage for state transition testing is measurable for different lengths of transition sequences. The state transition coverage measure is:



Chows n-switch coverage

where  $n = \text{sequential transitions} - 1$ .

We could also say that  $N = \text{no. of "in-between-states"}$ .

Chows n-switch coverage is the percentages of all transition sequences of

$n-1$  transitions' length tested in a test suite.

State transition testing coverage is measured for valid transitions only. Valid transitions are transitions described in the model. There may, however, also be invalid, or so-called null-transitions and these should be tested as well.

#### 4.1.5.2 State Transition Testing Templates

A number of tables are used to capture the test conditions during the analysis of state transition machines.

To obtain Chows 0-switch coverage, we need a table showing all single transitions. These transitions are test conditions and can be used directly as the basis for test cases. A simple transition table is shown next.

The fields in the table are:



*Test design item number:* Unique identifier of the test design item

*Traces:* References to the requirement(s) or other descriptions covered by this test design

*Assumptions:* Here any assumption must be documented

The table must have a column for each of the defined transitions (three are shown here). The information for each transition must be:

*Transition:* The identification of the transition

*Start state:* The identification of the start state (for this transition)

*Input:* The identification or description of the event that triggers the transition

*Expected output:* The identification or description of the action connected to the transition

*End state:* The identification of the end state (for this transition)

<b>Test design item number:</b>		<b>Traces:</b>	
<b>Assumptions:</b>			
<b>Transition</b>			
<b>Start state*</b>			
<b>Input</b>			
<b>Expected output</b>			
<b>End state*</b>			

\* the “start” and “end” states are for each specific transition (test condition) only, not the state machine

Testing to 100% Chows 0-switch coverage detects simple faults in transitions and outputs.

To achieve a higher Chows  $n$ -switch coverage we need to describe the sequences of transitions.

The table to capture test conditions for Chows 1-switch coverage is shown next.



Test design item number:	Traces:		
<b>Assumptions:</b>			
<b>Transition pair</b>			
<b>Start state*</b>			
<b>Input</b>			
<b>Expected output</b>			
<b>Intermediate state*</b>			
<b>Input</b>			
<b>Expected output</b>			
<b>End state*</b>			

Here we have to include the intermediate state and the input to cause the second transition in each sequence. Again we need a column for each set of two transitions in sequence.

If we want an even higher Chows n-switch coverage we must describe test conditions for longer sequences of transitions.



As mentioned above we should also *test invalid transitions*. To identify these we need to complete a state table. A state table is a matrix showing the relationships between all states and events, and resulting states and actions.

A template for a state table matrix is shown below. The matrix must have a row for each defined state and a column for each input (event). In the cross-cell the corresponding end state and actions must be given.

An invalid transaction is defined as a start state where the end state and action is not defined for a specific event. This should result in the system staying in the start state and no action or a null-action being performed, but since it is not specified we cannot know for sure.

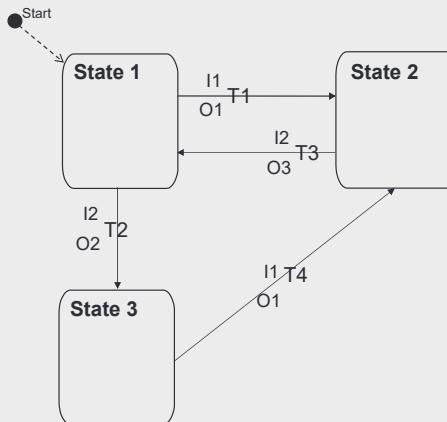
<b>Start state</b>	<b>Input</b>
	<b>End state / Action</b>

The “End state / Action” for invalid transitions must be given as the identification of the start state / “N” or the like.

A test condition can be identified from this table for each of the invalid transitions.

### 4.1.5.3 State Transition Testing Example

In this example we are going to identify test conditions and test cases for the state machine shown here.



I1 = Push button A

I2 = Push button B

O1 = Bib

O2 = Light on

O3 = Light off



*Don't worry about what the system is doing—that is not interesting from a testing point of view.*

The drawing of the state machine shows the identification of the states, the events (inputs), the actions (outputs), and the transitions. The descriptions of the inputs and outputs are given to the right of the drawing.

First of all we have to define test conditions for all single transitions to get Chows 0-switch coverage.

<b>Test design item number:</b> 2.4		<b>Traces:</b> State machine 1.1			
<b>Assumptions:</b> None					
Transition	T1	T2	T3	T4	
<b>Start state*</b>	S1	S1	S2	S3	
<b>Input</b>	I1	I2	I2	I1	
<b>Expected output</b>	O1	O2	O3	O1	
<b>End state*</b>	S2	S3	S1	S2	

Identification of sequences of two transitions to achieve Chows 1-switch coverage results in the following table.

<b>Test design item number:</b> 2.5		<b>Traces:</b> State machine 1.1				
<b>Assumptions:</b> None						
Transition pair		T1/T3	T1/T3	T3/T2	T2/T4	T4/T3
Start state*	S1	S2	S2	S1	S3	
Input	I1	I2	I2	I2	I1	
Expected output	O1	O3	O3	O2	O1	
Intermediate state*	S2	S1	S1	S3	S2	
Input	I2	I1	I2	I1	I2	
Expected output	O3	O1	O2	O1	O3	
End state*	S1	S2	S3	S2	S1	

We will not go further in sequences.

The next thing will be to identify invalid transitions. To do this we fill in the state table. The result is:

Start state	Input	I1	I2
S1		S2/O2	S3/O2
S2		S2/N	S1/O3
S3		S2/O1	S3/N

We have two invalid transitions:

State 2 + Input 1 and State 3 + Input 2.

The test cases to execute can now be created from these test conditions and written directly into a test procedure.

<b>Test procedure:</b> 3.5			
<b>Purpose:</b> This test procedure tests single valid and invalid transitions.			Traces: State machine 1.1
<b>Prerequisites:</b> The system is in State 1			
<b>Expected duration:</b> 5 minutes			
Tag	TC	Input	Expected output
T1	2	Reset to state 1 + push button B	The system bibs + state 2
T2		Reset to state 1 + push button B	The light is on + state 3
IV2		Push button B again	Nothing changes
T4		Push button A	The system bibs + state 2
IV1		Push button A again	Nothing changes
T3		Push button B	The light is off + state 1

#### 4.1.5.4 State Transition Testing Hints

We should try to avoid invalid transitions by defining the results of invalid events. This is called defensive design, and it is a design activity.

If it is not practical to define all possible state and event combinations explicitly, we should encourage the designers to define a default for truly invalid situations. It could for example be defined that all null-transitions should result in a warning.

One of the difficulties of state machine is that they can become extremely complex faster and more often than you imagine. State machines can be defined in several levels to keep the complexity down, but this is a design decision.

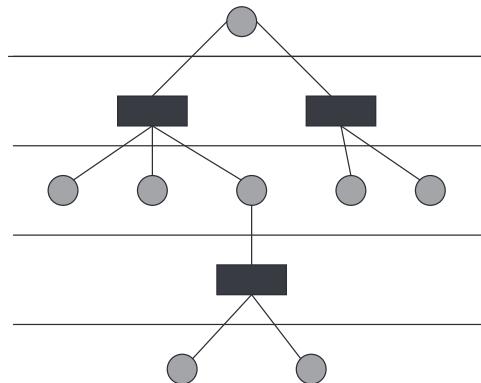


#### 4.1.6 Classification Tree Method

The classification tree method is a way to partition input and state domains into classes. The method is similar to equivalence partitioning, but can handle more complex situations where input or output domains can be looked at from more than one point of view.

The idea in the classification tree method is that we can partition a domain in several ways and that we can refine the partitions in a stepwise fashion. Each refinement is guided by a specific aspect or viewpoint on the domain at hand.

The result is a classification tree like the one shown here.



There are two types of nodes in the tree

- ▶ (Sub)domain nodes
- ▶ Aspect nodes

*The two types must always alternate.*



The domain  is the full collection of all possible inputs and states at any given level in the tree. The state is a very broad term here; it means anything that characterizes the product at a given point in time and includes for example which window is current, which field is current, and all data relevant for the behavior both present on the screen and stored “behind the screen.”

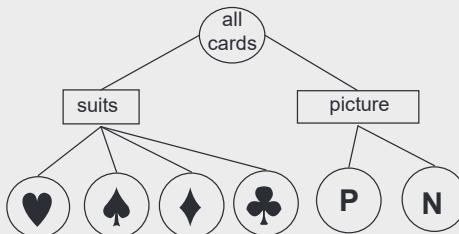
The aspect  is the point of view you use when you are performing a particular partitioning of the domain you are looking at. It is very important to be aware that it is possible to look at the same domain in different ways (with different aspects) and get different subdomains as the result. This is why there can be more aspects at the same level in the classification tree and more subdomains at the same level (under the aspects) as well.

### Ex.

An ordinary deck of cards (without jokers) can be viewed and hence partitioned in different ways. The aspects could be:

- ▶ Suit (spades, hearts, diamonds, clubs)
- ▶ Picture or number

The classification tree resulting from this analysis would look like this:



### Ex.

There are a few rules that need to be observed when we make the classification tree. Under a given aspect:

- ▶ Any member of the domain must fit into one and only one subdomain under an aspect—it must not be possible to place a member in two or more subdomains
- ▶ All the members of the domain must fit into a subdomain—no member must fall outside the subdomains

### Ex.

If we look at the “suit” aspect above, no card can be in two suits, and all cards belong to a suit.

When we create a classification tree we start at the root domain (which is highest in the graph!). This is always the full input and state domain for the item we want to examine. We must then:

- ▶ Look at the domain and decide on the views or aspects we want to use on the domain
- ▶ For each of these aspects
  - ▶ Partition the full root domain into classes. Each class is a subdomain.
  - ▶ For each subdomain (now a full domain in its own right)
    - ▶ Decide aspects that will result in a new partitioning
    - ▶ For each aspect
    - ▶ And so on

At a certain point it is no longer possible or sensible to apply aspects to a domain. This means that we have reached a leaf of the tree. The tree is finished when all our subdomains are leaves (the lowest in the graph!).

Leaves can be reached at different levels in the classification tree. The tree does not have to be symmetric or in any other way have a predictable shape.

A leaf in a classification tree is similar to a class in an equivalence partitioning: We only need to test one member, because they are all assumed to behave in the same way.



#### 4.1.6.1 Classification Tree Method Coverage

The coverage for a classification tree is the percentage of the total leaf classes tested in a test suite.

Leaves belonging to different aspects can be combined, so that we can reach a given coverage with fewer test cases. In areas of high risk we can also choose to test combinations of leaf classes.

#### 4.1.6.2 Classification Tree Method Test Design

##### Template

It is usually more practical to present a classification tree in a table rather than as a tree. A template for such a table where the test conditions are captured, is shown below.

Test design item number.:			Traces:				
Assumptions:							
Domain 1	Aspect 1	Domain n	Aspect n	Tag	Tc1		Ten



The fields in the table are:

*Test design item number*: Unique identifier of the test design item

*Traces*: References to the requirement(s) or other descriptions covered by this test design

*Assumptions*: Here any assumption must be documented.

*Domain 1*: A description of the (root) domain

*Aspect 1*: A list of the aspects defined for domain 1

For each aspect a list of subdomains are made.

For each of the subdomains new aspects are identified or the subdomain is left as a leaf.

This goes on until we have reached the leaves in all branches.

*Tag*: Unique identification of the leaves = test conditions

*Tc1*: A marking of which test cases cover the test conditions

#### 4.1.6.3 Classification Tree Method Example

**Ex.**

In this example we are going to test the following requirements for a small telephone book system:

- (1) A person can have more than one phone number
- (2) More than one person can have the same phone number
- (3) There is one input field where you can type either:

- ▶ Phone number
- ▶ All names in the full name
- ▶ Some of the names in the full name

- (4) A person shall be found if one or more names match
- (5) One or more people shall be found if the phone number matches
- (6) The output shall be either:

- ▶ An entry for each person that is found
- ▶ An error message: no person found

First we fill in the header:

<b>Test design item number:</b> 56	<b>Traces:</b> Req. (1) – (6)
<b>Assumptions:</b> None	

The next thing is to look at the root and identify the first level of aspects:

Domain 1	Aspect 1
All inputs and types of lists	
	<i>input type</i>
	<i>match?</i>
	<i>state of list</i>
	<i>result</i>

Now we must take each of these aspects one by one and filter the root domain through them. The leaf subdomains are marked in bold.

Domain 1	Aspect 1	Domain 2
All inputs and types of lists		
	<i>input type</i>	<b>pure text</b>
		<b>pure number</b>
		<b>mixture</b>
		<b>empty</b>
	<i>match?</i>	<b>no</b>
		yes
	<i>state of list</i>	<b>empty</b>
		not empty
	<i>result</i>	<b>nothing found</b>
		something found

We now have seven subdomains that are leaves, and three that can be further broken down by new aspect. So we find some aspects for each of the remaining subdomain, and find the subdomain for each.

The “old” leaves have been left out of the table for the time being, and the “new” leaves are marked in bold.

Domain 1	Aspect 1	Domain 2	Aspect 2	Domain 3
All inputs and types of lists				
	<i>match?</i>	yes	<i>no.of matches</i>	<b>1</b>
				<b>more than 1</b>
			<i>type of match</i>	name
				<b>phone no.</b>
	<i>state of list</i>	not empty	<i>1 name + 1 no.</i>	<b>yes</b>
				no

Domain 1	Aspect 1	Domain 2	Aspect 2	Domain 3
All inputs and types of lists				
	<i>match?</i>	yes	<i>no.of matches</i>	1
				<b>more than 1</b>
			<i>type of match</i>	name
				phone no.
	<i>state of list</i>	not empty	<i>1 name + 1 no.</i>	<b>yes</b>
				<b>no</b>
			<i>1 name + many no</i>	<b>yes</b>
				<b>no</b>
			<i>many names + 1 no.</i>	<b>yes</b>
				<b>no</b>
			<i>1 name + 1 no.</i>	<b>yes</b>
				<b>no</b>
	<i>result</i>	something found	<i>1 name + many no.</i>	<b>yes</b>
				<b>no</b>
			<i>many names + 1 no.</i>	<b>yes</b>
				<b>no</b>

Now we have only one subdomain, which is not a leaf left to deal with, namely a match of the name. The aspect to use here is how much of the name matches, and the subdomains are: "full name" and "part of name." These are leaves.

Before we go any further we have to control the partitioning. We need to ensure that any member of a domain fits into one and only one subdomain.

In this example we find that a string of blanks belongs to more than one subdomain, because a blank is considered to be text. To resolve that we require that pure text contains at least one letter.

Our test conditions are now defined, and we must go on defining our test data and test cases.

If we examine the classification tree we can see that we need a number of phone lists as our test data.

**Telephone list 1**

- ▶ Bo Hansen 4311
- ▶ Neil Smith 4210 4545
- ▶ John Raven 4545

**Telephone list 2**

- ▶ Bo Hansen 4311
- ▶ Neil Smith 4210 4545

**Telephone list 3**

- ▶ Bo Hansen 4311
- ▶ John Raven 4545

**Telephone list 4**

- ▶ Neil Smith 4210 4545

**Telephone list 5 is empty**

All the test conditions identified in the classification tree can be covered by eight test cases. We could trace the test cases to the test conditions, but since some of the test cases cover quite a few test conditions we mark the test cases that cover the conditions in the condition table here.

The test cases we can design are:

TC	Input =	Expected output
1	Telephone list 1 "Bo Hansen"	Bo Hansen 43-11
2	Telephone list 1 "4545"	Neil Smith 4545 John Raven 4545
3	Telephone list 1 "John 2"	No person found
4	Telephone list 1 "4210"	Neil Smith 4210, 4545
5	Telephone list 2 " "	No person found
6	Telephone list 3 "Raven"	John Raven 4545
7	Telephone list 4 "Bo Hansen"	No person found
8	Telephone list 5 "43"	No person found

These tables tend to get rather big, and it is normally a good idea to handle them in a spreadsheet. The full condition table is shown in Appendix 4A.



#### 4.1.6.4 Classification Tree Method Hints

The classification tree method facilitates the test case design when it is too complicated to make equivalence class partitioning.

This is mainly the case when input is composed of more parts, and the relations between the input parts rather than the individual parts determine the outcome.

The method is useful when input can be considered as a unit, for example when the input is given via a graphical user interface. In such forms there are no sequences in the input; the user determines the sequence and the input is treated when the user decides that the form is filled in.

In other types of user interfaces, where the sequence of the input is predetermined, equivalence partitioning can be used on each individual input item.



Remember that not only what is entered by the user is input. Lists, files, database tables, and other types of data used by the system are also part of the “input.”

#### 4.1.7 Pairwise Testing

When we make test cases from a classification tree, the combinations of the leafs we get in our test cases are often more or less selected at random. Furthermore we often do not get all possible combinations tested. This may also be the case when we are testing products with other types of possible combinations of configurations (preconditions) or of inputs.



An example of a product with a number of precondition possibilities is a system that may run:

- ▶ On three different browsers
- ▶ Using two different database administration systems
- ▶ On four different operating systems

This system has limited possibilities when you think about it, but even so there are  $3 \times 2 \times 4 = 24$  different possible combinations to test.



Another example is a system with a form for entering different information about clients, both actual and potential. The following information must be supplied: the values in brackets after the information type are the possible valid values to select from:

- ▶ Size (small, medium, large)
- ▶ Business (private, civil administration, defense)
- ▶ Relevance (low, middle, high)
- ▶ Status (customer, lead, potential)

Here we have  $3 \times 3 \times 3 \times 3 = 81$  possible combinations.

In some cases it may be possible and relevant to test all combinations of preconditions and/or inputs. But in some cases it is not, and then we have to have another way of designing sufficient test cases.

The pairwise test case design technique is about testing pairs of possible combinations. This reduces the number of test cases compared to testing all combinations, and experience shows that it is sufficiently effective in finding defects in most cases.

It is not always an easy task to identify all the possible pairs we can make from the combination possibilities. There are two different techniques to assist us in that task, namely orthogonal arrays and the allpairs algorithm. There is not objective evidence as to which technique is the best, but both techniques have their fans and their opponents.

#### 4.1.7.1 Pairwise Testing Coverage

It is possible to measure the coverage of all pairs. It is simply measured as the percentage of the possible pairs that have been exercised by a test.

#### 4.1.7.2 Orthogonal Arrays

Orthogonal arrays were first described by the Swiss mathematician Leonhard Euler, born in 1707. He introduced much of the modern terminology for mathematical analysis, including the notation for mathematical functions.

An orthogonal array is a two-dimensional array (a matrix) of values ordered in such a way that all pairwise combinations of the values are present in any two columns of the arrays.

1	1	1
1	2	2
2	1	2
2	2	1

An example of an orthogonal array is:

Select any two columns and all the possible pairs of 1 and 2:  
 $(1,1)$ ;  $(1,2)$ ;  $(2,1)$ ; and  $(2,2)$   
 are present.

Orthogonal arrays are said to be balanced, because the number of times one possible pair is present, all the pairs will be present the same number of times (in the simple example 1 time).

An orthogonal array is mixed if not all the columns have the same range of values. We can have an orthogonal array where one column only has 1s and 2s and other columns have 1s, 2s, and 3s, for example.

The size and contents of orthogonal arrays are usually described in a general manner like:

$(N, s1k1 s2k2 \dots, t)$



Ex.



where

$N$  = number of rows (or runs)

$s$  = number of levels = number of different values

$k$  = number of factors = number of columns for the corresponding  $s$

$t$  = strength = in any  $t$  columns you see each of the  $st$  possibilities equally often ( $t$  is usually 2 and in that case often omitted from the description)

Note that the description is often ordered so that the  $s$ 's are ordered in ascending order, though the actual columns in the array may be arranged differently.

This notation is very helpful when we are looking for suitable orthogonal arrays for our testing task.

### Ex.

The simple orthogonal array shown above can be described as: (4, 23)

An orthogonal array described as (72, 25 33 41 67) is a mixed array with 72 rows, 5 columns with 2 different values, 3 columns with 3 different values, 1 column with 4 different values, and 7 columns with 6 different values, that is 16 columns in all.



Creating orthogonal arrays is not a simple task. Many people have contributed to libraries of orthogonal arrays, and new ones are still being created. A large number of arrays in all sizes and mixtures may be found on [www.research.att.com/~njas/oadir/index.html](http://www.research.att.com/~njas/oadir/index.html).

We can use orthogonal arrays to help us identify all the pairs of possible inputs or preconditions that we want to test. What we need to do is find a suitable array and substitute the values in this with our values. If we then design test cases corresponding to each row, we are guaranteed to have tested all the possible pairs.



The process is the following:

- ▶ Identify the inputs/preconditions (IPs) that can be combined
- ▶ For each of the IPs find and count the possible values it can have (e.g., (IP1; $n=2$ ); (IP2; $n=4$ ) and so on)
- ▶ Find out how many occurrences you have of each  $n$ , (e.g., 3 times  $n=2$ , 1 times  $n=4$  and so on) (this provides you with the needed sets of  $sk$  (e.g., 23 41))
- ▶ Find an orthogonal array that has a description of at least what you need—if you cannot find a precise match, take a bigger array; this often happens, especially if we need a mixed array
- ▶ Substitute the possible values of each of the IPs with the values in the orthogonal array—if we had had to choose an array that was too big, we could just fill in the superfluous cells with valid values chosen at random
- ▶ Design test cases corresponding to each row in the orthogonal array

### *Orthogonal Array Example*

This example covers the system with the input possibilities and their corresponding valid values shown here:

**Ex.**

- ▶ Size (small, medium, large)
- ▶ Business (private, civil administration, defense)
- ▶ Relevance (low, middle, high)
- ▶ Status (customer, lead, potential)

There are four input possibilities, and each of them has three possibilities, so we need at least an array of (34). One such array can be found on the Internet, namely:

1	1	1	1
1	2	2	3
1	3	3	2
2	1	2	2
2	2	3	1
2	3	1	3
3	1	3	3
3	2	1	2
3	3	2	1

We will now assign the first column to size, and substitute the values in the array with the possible values for size.

The array will look as shown below, where we have added an extra row to show which column represents which input:

size	1	1	1
small	2	2	3
small	3	3	2
small	1	2	2
medium	1	2	2
medium	3	1	3
medium	3	1	3
large	1	3	3
large	2	1	2
large	3	2	1

The fully substituted orthogonal array is shown below:

size	business	relevance	status
small	private	low	customer
small	civil administration	middle	potential
small	defense	high	lead
medium	private	middle	lead
medium	civil administration	high	customer
medium	defense	low	potential
large	private	high	potential
large	civil administration	low	lead
large	defense	middle	customer

From this table we can design the nine low-level test cases needed to get 100% pair coverage by using the values given in each row as the input values for our four fields.

We still have to derive the expected results from the requirements or other basis material—as usual the test case design technique can not provide those.

#### 4.1.7.3 Allpairs Algorithm

James Bach has created: “a script which constructs a reasonably small set of test cases that include all pairings of each value of each of a set of parameters.” The script is called Allpairs and it is available from James Bach’s Web site at [www.satisfice.com](http://www.satisfice.com).

The principle of finding the pairs is different from using orthogonal arrays, but the aim is the same: to reduce the number of test cases to run when testing combinations of a number of inputs/preconditions each with a number of valid values.

In the words of James Bach: “The Allpairs script does not produce an optimal solution, but it is good enough.” This is, of course, James Bach’s opinion and experience, and it is not necessarily true in all contexts.

#### 4.1.7.4 Higher-Order Combinations

The techniques described above are concerned with getting pairs of possible values for test cases. We can also choose to test higher-order combinations, such as triples or more, but that is rarely done.

#### 4.1.7.5 Pairwise Testing Hints

When we create pairs of values as described earlier, all the values have equal weight. This means that neither orthogonal arrays nor the Allpairs algorithm takes the distribution of the values and the risks associated with individual pairs into account.

It may well be, that one particular value is way more common than any of the others—no doubt for example that there are much more private businesses than defense-related businesses around.

It may also be possible that a specific combination has a much larger risk level than the others (i.e., that the effect if a defect is not found around that combination, is much higher than for all other combinations), it could, for example, be serious for our company if all defense/leads were left out of a mailing list with invitations to a special sales event, whereas it would hardly make any difference if small/private were missed from a general mailing campaign.

To overcome this we should supply pairwise testing with risk analysis and design more test cases around the combinations with a high risk level.

In some cases the pairs we have established to test are not actually testable. One value of one input may be prohibitive for a specific value for another input. In this case there is nothing else to do than leave that test case out and explain why a lower coverage than expected was achieved.



#### 4.1.8 Use Case Testing

The concept of use cases was first developed by the Swedish Ivar Jacobson in 1992 and has since made triumphal progress in the IT development world.

A use case or scenario as it is also called shows how the product interacts with one or more actors. It includes a number of actions performed by the product as results of triggers from the actor(s). An actor may be a user or another product with which the product in question has an interface.

Use cases are much used to express user requirements at an early state in the development and they are therefore excellent as a basis for acceptance testing (if they are kept up-to-date and still reflect the product as it has turned out in the end!).

Use cases should be presented in a structured textual form with a number of headings for which the relevant information must be supplied. There are many ways of structuring a use case, and it is up to each organization to define its own standard.



The following example shows a set of headings for a use case with the explanations of what to write under each.

<b>Use case:</b>	The name of the use case. The name should be as descriptive as possible.
<b>Purpose:</b>	The goal of the use case (i.e., what is achieved when it is completed).
<b>Actor:</b>	Who (in terms of a predefined role) is interacting with the product.
<b>Preconditions:</b>	Any prerequisite that must be fulfilled before the use case may be performed.
<b>Description:</b>	The following is a list of actions to be performed. The list should have no more than 20 steps (n); otherwise the use case should be divided.
<b>Actor</b>	<b>Product</b>
1.	
	n.
<b>Postconditions:</b>	The state in which the product and/or the actor is to be found in when the use case has been performed.
<b>Variants and exceptions:</b>	Any specific cases in terms of variants or exceptions must be described here. This part often constitutes the bulk of the use case.
<b>Rules:</b>	Any specific rules or complex calculations are described here, or references are made to, for example, standards where such issues are detailed.
<b>Safety:</b>	Any safety considerations are described here.
<b>Frequency:</b>	How often the use case must be performed.
<b>Critical conditions:</b>	Any conditions under which the performance of the task is especially critical (e.g., in terms of response time or volume).

Testing a use case involves testing the main flow as specified in the steps in the description. Depending on the associated risks it may also include testing the variants and exceptions.

Note that the description of the main flow is usually much shorter than the descriptions of the variants and exceptions; these may also be considerably more difficult to test, if at all possible.

As it can be seen in the template above a good use case provides a lot of useful information for testing purposes. It should in fact be possible to design our test procedures directly from the use case description.

We can get the identification of the use case for traceability purposes and



the necessary preconditions directly from the form. The high-level test cases can be extracted directly from the steps in the description, where it should be ensured that each step provides the preconditions for the next one, except for the last which provides the expected postconditions.

A use case description will rarely contain actual input values; these must be selected when we design our low-level test cases. Appropriate specification-based techniques (for example, equivalence partitioning, boundary value analysis, and pairwise testing) may be used to select the actual values to use.

Based on the description, the postconditions, and possibly the rules it should be possible to derive expected results for each test case.

The information given for variants and exceptions, safety, frequency, and critical conditions can be used for risk analysis and decisions about which variants and exceptions to test to which depths.



#### 4.1.8.1 Use Case Testing Coverage

Since a use case is not something easily measurable, there is no coverage item defined for use case testing, and it is therefore not possible to determine the coverage.

#### 4.1.9 Syntax Testing

*Note: This technique is not part of the ISTQB syllabus, but included here because I find it very useful.*



Syntax is a set of rules, each defining the possible ways of producing a string in terms of sequences of, iterations of, or selections among other strings.

Syntax is defined for input to eliminate “garbage in.”

Many of the “strings” we are surrounded by in daily life are guided by syntax.

For example



- ▶ Web addresses: www.aaaaa.aa, aaaaa.aa  
(note the difference in appearances! My word processor recognizes the first address, but not the second address, because the www is missing)
- ▶ CPR number: ddmmyy-nnnn  
(Danish Central Person Registration number)
- ▶ Credit card number: nnnn nnnn nnnn nnnn  
(my VISA card—other cards have other syntaxes)

In the examples above the rules for the different strings are expressed using for example “n” to mean that a number should be at a specific place in the string or “dd” to indicate a day number in a month.

We can set up a list of rules, defining strings as building blocks and defining a notation to express the rules applied to the building blocks in a precise and compressed way. The building blocks are usually called the elements of the entire string.

The syntax rule for the string we are defining must be given a name.

The most commonly used notation form is the Backus-Naur form. This form defines the following notations

""	elementary part, e.g.	"1" "z"
	alternative separator	"A"   "B"
[ ]	optional item(s)	[ " " ]
{}	iterated item	

These notations can be used to form elements and the entire string.

### Ex.

An example of a syntax rule for a string called pno. could be:

pno. = 2d [ " " ] 2d [ " " ] 2d [ " " ] 2d

Here we have defined the elements:

2d = dig dig

dig = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

This means that the pno. string must consist of four sets of two digits. The digits can range from 0 to 9. The sets of two digits can be separated by blanks, but they can also not be separated.

A valid string following this syntax is a Danish telephone number:  
39 62 36 48.

The way my father used to write his telephone number is, however, illegal according to this syntax: 45 940 941.



To derive test conditions we need to identify options in the syntax and test these independently. Options appear when we can choose between elementary parts or elements for a given element or for the entire string. Syntax testing does not include combinations of options as part of the technique.

There is no coverage measure for syntax testing.

To make a negative test we need to test invalid syntax as well. For this we operate with possible mutations. Examples of the most common mutations are:

- ▶ Invalid value is used for an element
- ▶ One element is substituted with another defined element
- ▶ A defined element is left out
- ▶ An extra element is added



#### 4.1.9.1 Syntax Testing Templates

The design of the test conditions based on syntax can be captured in a table like the one shown here. The fields are the standard fields in test condition templates, as described earlier.

<b>Test design item number:</b>	<b>Traces:</b>
<b>Based on: Input / Output</b>	<b>Assumptions:</b>
<b>Tag</b>	<b>Description</b>

#### 4.1.9.2 Syntax Testing Example

In this example we will test the input of a member number. The syntax defined for the member number is

member no. = type "no" "mm"- "yy

First we list the options derived from the entire string, the elements, and the elementary parts.

**Ex.**

<b>options</b>	
member no. = type "no" "mm"- "yy	none
type = "B"   "S"   "G"	3
no = dig dig dig	none
mm = "01"   "02"   .....   "11"   "12"	12
yy = dig dig	none
dig = "0"   "1"   "2"   ...   "8"   "9"	10
	<u>25</u>

We have 25 possible independent mutations. We can list them in the template like this.

<b>Tag</b>	<b>Description</b>
T1	"B"   "S"   "G"
M1	"01"   "02"   .....   "11"   "12"
D1	"0"   "1"   "2"   ...   "8"   "9"

Since we have not included specifications of what is happening when a string with a valid syntax is entered (nor what happens if the string is invalid) we will only list the inputs for the test cases we can design from the test conditions.

The following table shows a few of the 25 possible test cases.

<b>TC</b>	<b>Tag</b>	<b>Input</b>
TC1	T1	B 326 04-05
TC2	T1	S 326 04-05
TC4	M1	G 326 01-05
TC5	M1	G 326 02-05
TC6	M1	G 326 03-05
TC15	M1	G 326 12-05
TC16	M1	G 111 01-11
TC17	M1	G 222 01-22
TC24	M1	G 999 01-99
TC25	M1	G 000 01-00

The number of test cases may be reduced by having a single test case cover several options. This may, however, reduce the fault correction time, if failures are encountered, because it can be more difficult to locate the fault.

To test *invalid syntax* we list the mutations we want to try. These are:

<b>Tag</b>	<b>Mutation description</b>
MU1	Invalid value – Applicable to all positions in the string
MU2	Substitute – Any two elements
MU3	Element missing – Applicable to all elements
MU4	Extra element – Anything, but may not be possible

There is an infinite number of possible test cases for the mutations. We will only list a few here:

TC	Tag	Input
TC1	MU1	F 456 02-99
TC2	MU1	B-326 02-99
TC3	MU1	B a26 02-99
TC4	MU1	B-326 02-9g
TC12	MU2	BB456 02-99
TC15	MU2	B B 02-99
TC33	MU3	B 02-99
TC34	MU3	B 456-99
TC62	MU4	BB 456 02-99

#### 4.1.9.3 Syntax Testing Hints

The number of invalid strings to test depends on the risk related to invalid input wrongly being accepted.

It can be a bit tricky to work with mutations, because some may be indistinguishable from correctly formed inputs if elements are identical. Some mutations may also be indistinguishable from each other, in which case they should be treated as one.

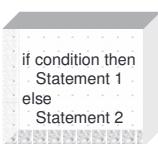
It is possible to define more mutations than those listed above, depending on the nature of the syntax.

To get an even stricter test we can use combinations of mutations. Who knows? Maybe two wrong elements at a time will cause the string to be accepted.

## 4.2 Structure-Based Techniques

The structural test case design techniques are used to design test cases based on an analysis of the internal structure of the component or system. These techniques are also known as white-box tests.

Traditionally the internal structure has been interpreted as the structure of the code. These techniques therefore focus on the testing of code and they are primarily used for component testing and low-level integration testing. In newer testing literature, structural testing is also applied to architecture where the structure may be a call tree, a menu structure, or a Webpage structure.





The structural test case design techniques covered here are:

- ▶ Statement testing
- ▶ Decision testing/branch testing
- ▶ Condition testing
- ▶ Multiple condition testing
- ▶ Condition determination testing
- ▶ LCSAJ (loop testing)
- ▶ Path testing
- ▶ (Interunit testing—not part of the ISTQB syllabus)

The test case design techniques in this category all require that the tester understands the structure (i.e., has some knowledge of the coding language). The tester doesn't necessarily need to be able to write code. It is like with a foreign language: You may be able to understand what is being said, but find it difficult to express yourself. This is usually not a problem anyway, because most structural testing of code is performed by programmers.

Structural testing is very often supported by tools, because the execution of components in isolation requires the use of stubs and drivers.

The test coverage can be measured for the structural test techniques. Test cases are designed to get the required coverage for the specified coverage item. If the coverage is expressed as statement coverage for example, the input to the test cases is determined using the statement test case design technique.



It is generally a good idea to start any test with specification-based, defect-based, and experience-based testing and measure the coverage achieved by executing tests designed with these techniques. This is cheaper and easier than to start off with structure-based techniques. Only if the achieved coverage is too low, should the appropriate structure-based technique be brought into action.

The techniques provide us with ideas for input. For the low-level test cases the concrete input values are selected. Subsequently the expected output is determined.

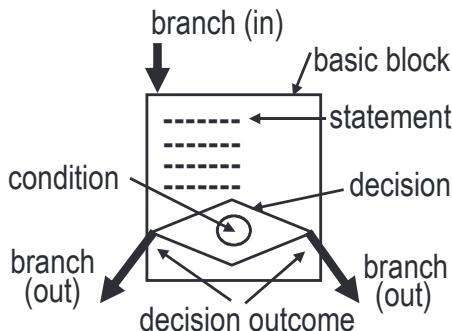


From where do we derive the expected output?

*From the requirements—NEVER, ever from the code!*

#### 4.2.1 White-Box Concepts

Before we go any further there are some white-box concepts that need to be defined. They are illustrated in the “white-box inset” shown on the opposite page.



The first concept is that of a *statement*. An executable statement is defined as a noncomment or nonwhite space entity in a programming language, typically the smallest indivisible unit of execution.

A group of statements always executed together—or not at all—is called a *basic block*. A basic block can consist of only one statement or it can consist of many. There is no theoretical upper limit for the number of statements in a basic block.

The last statement in a basic block will always be a statement that leads to another building block, or stops the execution of the component we are working with (e.g., return) or the entire software system (end).

The most interesting is, however, when a basic block ends in a decision, that is a statement where the further flow depends on the outcome of the decision. Decision statements are for example IF ... THEN ...ELSE, FOR ..., DO WHILE..., and CASE OF....

A decision statement is also called a *branch point*. A *branch* is a (virtual) connection between basic blocks. One or more branches will lead into a basic block (except to the first), and likewise there will be one or more branches leading out of a basic block (except the last).

The branches out of a basic block are connected to the outcomes of the decision, also called *decision outcome* or branch outcome. Most decisions have two outcomes (True or False), but some have more, for example Case statements.

The last concept is that of a condition. A *condition* is a logical expression that can be evaluated to either True or False. A decision may consist of one simple condition, or a number of combined conditions.

## 4.2.2 Statement Testing

Statement testing is a test case design technique in which test cases are designed to execute statements.

A statement is executed in its entirety or not at all.



$b = 3 + a;$  is one statement, whereas

if  $a = 2$  then  $b = 3 + a$  end if; is more than one statement!

The definition of a statement is independent of how the code is actually written and what language it is written in.



In statement testing we design test cases to get a specifically required *statement coverage*. Statement coverage is the percentage of executable statements (in a component) that have been exercised by the test. Statement coverage is the weakest completion criterion we can have.

The component under test must be decomposed into the constituent statements, and we derive input for test cases from the code.

### Ex.

In the first example we have this small piece of code:

```
Read A;
Read B;
if A = 245 then
    Write 'Bingo';
endif;
Write B;
```

There are five statements (since “endif” doesn’t count).

To get 100% statement coverage we need one test case:

TC	Input	Expected output
TC1	A = 245	

Note that we have no means of finding out what the expected output is, because the corresponding requirements (or design) are not included.

### Ex.

In the next example we will use this piece of code:

```
Read A;
Read B;
if A = 245 then
    Write 'Bingo';
endif;
if A < B then
    A = B;
else
    A = 0;
endif;
```

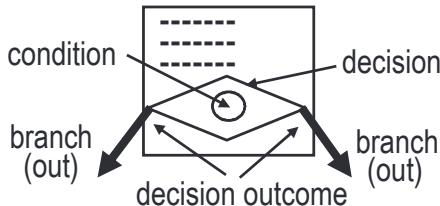
Here we have eight statements. To get 100% statement coverage we need two test cases:

TC	Input	Expected output
TC1	A = 245, B = 250	
TC2	A = 400, B = 250	

Again, we don't know what the expected results may be.

### 4.2.3 Decision/Branch Testing

Decision and branch testing have coexisted for many years. Experience has shown that it may be quite difficult to define branches correctly, whereas decisions and decision outcomes are much easier to define.



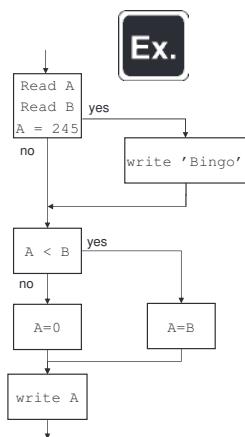
At 100% coverage branch coverage and decision coverage give identical results.

We can also see that decision outcome is defined to be equal to branch outcome, namely (according to BS7925-1): The result of a decision (which therefore determines the control flow alternative taken).

To define test cases for decision testing we have to:

- ▶ Divide the code into basic blocks,
- ▶ Identify the decisions (and hence the decision outcomes and the branches)
- ▶ Design test cases to cover the decision outcomes or branches

In most cases a decision has two outcomes (True or False), but it is possible for a decision to have more outcomes, for example, in "case of ..." statements.



Let us look at the write “Bingo” example again. The flow diagram corresponding to the code is shown here. We can see that this code has seven branches and four decision outcomes.

First we set  $A = 240$  and  $B = 120$ .

This input covers three branches and we get branch coverage  $= 3/7 = 43\%$ . It also covers four decision outcomes, and we get a decision outcome coverage of  $2/4 = 50\%$ .

Next we set  $A = 245$  and  $B = 360$ .

This covers four more branches and two more decision outcomes, and we have now got 100% branch coverage and 100% decision outcome coverage.

It (often) requires more test cases to obtain 100% branch and decision outcome coverage than to obtain 100% statement coverage.

Decision coverage is usually measured using a software tool. Some tools can show the code and mark covered and uncovered decision outcomes by coloration of the code lines.



#### 4.2.3.1 Other Decisions

Decisions may also be achieved by other statements than those using Boolean conditions, for example “case,” “switch,” or “computed goto” statements, or counting loops (implemented by “for” or “do” loops).

These should not go by untested. We have two options available for designing test cases for these, namely:

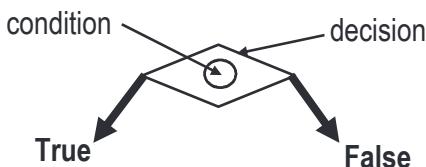
- ▶ Assume that the decision is actually implemented as an equivalent set of Boolean conditions
- ▶ Use a condition testing test case design technique as a supplement to decision testing

#### 4.2.4 Condition Testing

A condition is a Boolean expression containing no Boolean operators, such as AND or OR. A condition is something that can be evaluated to be either TRUE or FALSE, like: “ $a < c$ .”

A statement like “X OR Y” is not a condition, because OR is a Boolean operator and X and Y Boolean operands. X and Y may in themselves be conditions.

Conditions are found in decision statements.



Decision statements may have one condition like:

if ( $a < c$ ) then ...

They may also be composed of more conditions combined by Boolean operands, like:

if ( $(a=5)$  or ( $(c>d)$  and ( $c < f$ ))) then ...

or for short: if ( $X$  or ( $Y$  and  $Z$ )) then ...

The condition outcome is the evaluation of a condition to be either TRUE or FALSE. In condition testing we test condition outcomes.

The *condition coverage* is the percentage of condition outcomes in every decision (in a component) that have been exercised by the test.

So to get 100% condition outcome coverage we need to get each condition to be True and False (i.e., two test cases).



In the first example with ( $a < c$ ) we can design the test cases:

Test case	a	c	Outcome
1	5	7	True
2	6	2	False



In the next, more complex, example with ( $X$  or ( $Y$  and  $Z$ )) we also need to get each of the condition to be True and False.

Without going into detail about how to get  $X$ ,  $Y$ , and  $Z$  to become True and False we can see, that we can still get 100% condition coverage with 2 test cases, namely for example:

Test case	X	Y	Z
1	True	True	True
2	False	False	False



A 100% condition coverage for a decision can usually be covered with two test cases regardless of the complexity of the decision statement.

The 100% condition coverage may even be achieved without getting a 100% decision outcome if the entire decision evaluates to the same for both cases! Condition testing may therefore be weaker than decision testing.

Condition testing and the condition coverage test measurement are vulnerable to Boolean expressions, which actually control decisions, being placed outside of the actual decision statement.



**Ex.**

Consider this example:

```
FLAG := A or (B and C);
if FLAG then
    do_something;
else
    do_something_else;
end if;
```

It may look as if we get 100% condition coverage by getting FLAG to evaluate to True and False, but in reality we need all of A, B, and C to evaluate to True and False.

To combat this we should design test cases for all Boolean expressions, not just those used directly in control flow decisions.

#### 4.2.5 Multiple Condition Testing

With this test technique we test combinations of condition outcomes. To get 100% multiple combination coverage we must test all combinations of outcomes of all conditions.

Because there are two possible outcomes for each condition (True and False) it requires  $2^n$  test cases, where  $n$  is the number of conditions, to get 100% coverage.

**Ex.**

If we take the example from above:

if (X or (Y and Z)) then ..  
we have three conditions. We therefore need  $2^3 = 8$  test cases to get 100% multiple condition coverage.

The test cases we need are:

Test case	X	Y	Z
1	True	True	True
2	False	True	True
3	False	False	True
4	False	False	False
5	False	True	False
6	True	False	False
7	True	True	False
8	True	False	True

The number of test cases grows exponentially with the number of conditions! This is a very thorough test, even though it may happen that some of the test cases are impossible to execute.

Sometimes the concept of “optimized expressions” jeopardizes the measurement of this test technique. Optimized expressions mean that a compiler short-circuits the evaluation of Boolean operators. In for example the programming language C the Boolean “AND” is always short-circuited: the second operand will not be evaluated when the outcome can be determined from the first operand.

Short circuits present no obstacle to branch condition coverage or condition determination coverage, but there may be situations where it is not possible to verify multiple condition coverage.

#### 4.2.6 Condition Determination Testing

Sometimes testing to 100% multiple condition coverage would be to go overboard in relation to the risk associated with the component.

In these cases we can use the condition determination testing technique. With this technique we should design test cases to execute branch condition outcomes that independently affect a decision outcome. This is a pragmatic compromise where we discard the conditions that do not affect the outcome.

The number of test cases needed to achieve 100% condition determination coverage depends on how the conditions are combined in the decision statements:

- ▶ As a minimum we need  $n+1$  test cases
- ▶ As a maximum we need  $2^n$  test cases

where  $n$  is the number of conditions.

Still working with the same example as before:

if (X or (Y and Z)) then ..  
we need the test cases listed in this table.

**Ex.**

Test case	X	Y	Z
1	True	-	-
2	False	True	True
3	False	False	-
4	False	True	False

A “-” means that we don't care about what the outcome is, because it does not have impact on the result.

We can hence get 100% modified condition decision coverage or condition determination coverage with just four test cases.

#### 4.2.7 LCSAJ (Loop Testing)

Sometimes a program needs to do the same thing a number of times with different values. Instead of having to repeat the same piece of code several times, coding languages allow loops, that is repetitive execution of the same statements with different values for the variables.

The exact decision statements that form loops depend on the coding language. The statements in a loop are called the loop body. For some types of loops the loop body will always be executed at least once; for others it may be skipped altogether depending on the conditions of the looping.


**Ex.**

```
Read A;
B = 0;
while A <= 245 do
    B = B + 1;
    A = A + 1;
end while;
Write A;
Write B;
```

This little bit of code shows a so-called while loop.

The loop is executed as long as the value of A is less than or equal to 245, and for each loop the value of both A and B will be augmented by 1.



LCSAJ testing is a test case design technique where loops are identified and test cases developed to test linear sequences of code that start at a specific point in the code and end with a jump (or at the end of the component). Such a sequence is called a LCSAJ (Linear Code Sequence And Jump). It may also be called a DD-Path (Decision-to-Decision Paths).

An LCSAJ is defined by

- ▶ The start of the linear sequence of executable statements
- ▶ The end of the linear sequence
- ▶ The target line to which the control flow is transferred at the end of the linear sequence

The three items in an LCSAJ are usually identified by their line numbers in the source code listing.

An LCSAJ starts either from the start of a component or from a point to which control flow may jump from other than the preceding line. An LCSAJ terminates either by a specific control flow jump or by the end of the component. LCSAJs may go forwards in the code, or they may go backwards if the start point is somewhere down the code and the end point at a higher point.

LCSAJ coverage is the percentage of LCSAJs in a component that are exercised by the test.



To design the test cases using this technique we must:

- ▶ Identify each code line by its number
- ▶ List the branches (maybe with a note of the necessary conditions to satisfy them)
- ▶ From this list, find the LCSAJ start points
- ▶ Derive the LCSAJ from each of the start points

For each LCSAJ and thereby possible test case we therefore identify the start line, the end line, and the target line for the jump.

Sometimes some reformatting of the code may be needed if the coding standard used does not support this technique. The basic formatting rule that must be observed is that each branch has to leave from the end of a line and arrive at the start of a line.



This LCSAJ example is taken from BS-7925. We will base it on the component shown here.

1. READ (Num);
2. WHILE NOT End of File DO
3.     Prime := TRUE;
4.     FOR Factor := 2 TO Num DIV 2 DO
5.         IF Num - (Num DIV Factor)\*Factor = 0 THEN
6.             WRITE (Factor, ' is a factor of', Num);
7.         Prime := FALSE;
8.         ENDIF;
9.     ENDFOR;
10.    IF Prime = TRUE THEN
11.         WRITE (Num, ' is prime');
12.         ENDIF;
13.     READ (Num);
14. ENDWHILE;
15. WRITE ('End of prime number program');



From  
BS-7925

The code lines have been defined by line numbers. The next step is to list the branches.

Branch	Type	Condition
(2 -> 3)		Requires not end of file
(2 -> 15)	: Jump	Requires end of file
(4 -> 5)		Requires the loop to execute
(4 -> 10)	: Jump	Requires the loop to be a zero-trip
(5 -> 6)		Requires the if in line 5 to be true
(5 -> 9)	: Jump	Requires the if in line 5 to be false
(9 -> 5)	: Jump	Requires a further iteration of the for loop
(9 -> 10)		Requires the for loop to have exhausted
(10 -> 11)		Requires prime to be true
(10 -> 13)	: Jump	Requires prime to be false
(14 -> 2)	: Jump	Always has to take place

The start points we can identify are the lines: 1, 2, 5, 9, 10, 13, and 15. The start points are sorted here; not listed in the order they are found in the table.

We must now derive the LCSAJs from each of the start points. With start point in line 1, we get the following LCSAJs:

(1, 2, 15) (1, 4, 10) (1, 5, 9) (1, 9, 5) (1, 10, 13) (1, 14, 2)

We will have to work our way through the list of starting points finding all the LCSAJs. The last one is (15, 15, exit).

When we design test cases to execute we must cover enough LCSAJs to get the coverage we want. It will almost always be so that each test case covers a number of LCSAJs.

It can also happen that some LCSAJs are impossible to execute. This must be handled when defining the completion criteria based on LCSAJ coverage.

 There are a number of classic pitfalls connected with loops. One defect to watch out for in loops is the creation of an infinite loop, that is the case where a defect causes the loop to (theoretically) continue looping for ever. Examine the loop shown above and imagine what would happen if the statement "A = A + 1;" was omitted in the loop. This would be evident in such a small piece of code, but loops may be quite long and complicated, and infinite loops are seen now and again.

Another issue is the sequence of the statements in the loop body. There is almost always an issue about doing something the first time around and/or the last time around in a loop. Again consider the example above: Will the result be the same if B is set to 1 before the loop is started? And what if the condition is "<" instead of "<="?

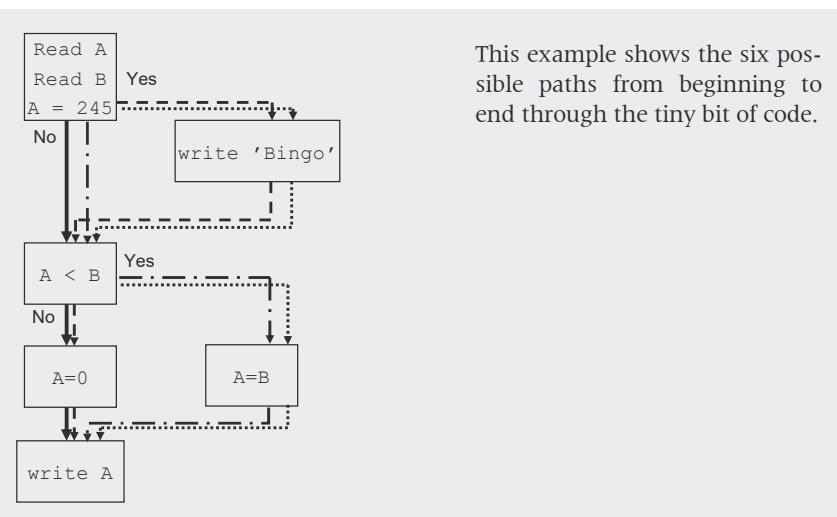
### 4.2.8 Path Testing

A path is a sequence of executable statements in a component from an entry point to an exit point.

Path coverage is the percentage of paths in a component exercised by the test cases.

When you execute test cases based on any of the other structure-based techniques described above, you will inevitably execute paths through the code.

This example shows the six possible paths from beginning to end through the tiny bit of code.



The number of possible paths through a component is exponentially linked to the number of decisions, especially decisions involving loops.

The control flow diagram shown here for a bit of code includes a loop that may be exercised up to 20 times. Such a loop only gives

$20 + 19 + \dots + 1$  possible different paths through the code.



In practice it may easily become impossible to obtain 100% path coverage when loops are involved.



Path testing includes a bit of error guessing. Experience shows that it can be useful to test these types of paths:

- ▶ Minimum path
- ▶ Path with no execution of any loop(s)
- ▶ Minimum path + one loop once
- ▶ Path with one loop a number of times
- ▶ Path with one loop the maximum number of times

where possible.

#### 4.2.9 Intercomponent Testing



*Note: This technique is not part of the ISTQB syllabus, but included here because I find it is very useful—especially since integration testing often is overseen and difficult to come to grips with.*

The idea in structured design, object-oriented design, and most other design paradigms is that the functionality is distributed on a number of components and/or systems for ease of production and maintenance.

This means that interfaces exist between interacting components and systems. At the component level we say that components call functions in each other, or, as it is expressed in object-oriented design, use methods that other classes make available. At the system level other types of interfaces exist, such as software-to-hardware integration, software-to-network integration, or software-to-manual procedure integration.

In the following the concept of function calls will be used, but the idea is exactly the same for method usage, and system interfaces.



The intercomponent testing technique is used in integration testing where the test objects are these interfaces. The integration comes after the component or system testing and assumes that logical and other types of defects in the body of the component or system have already been found.

The intercomponent testing is based on the design of the interfaces. For each function we should be able to find a design description of:

- ▶ Input: The input parameters required by the function
- ▶ Functionality (which we are not interested in at this point)
- ▶ Output: The resulting output parameters produced by the function



What we need to test are the calls to the functions, that is the interfaces. The coverage element is calls, and the intercomponent coverage is the percentage of the total number of calls that we have covered in a specific integration test.

To identify the total number of calls made by all the components or systems being integrated, we have to count, from the design, how many times each function is called. This is called fan-in: number of calls of a specific function from other functions (or the main program). This may be calculated using a static analysis tool; see Section 9.3.4. The sum of all the fan-ins provides the total number of calls.



For software components it is very difficult to say anything in general about the values for fan-ins. In system integration we normally have a very low fan-in, often only one, for each system.

Test cases are designed to cover the calls.

We may combine the intercomponent testing with other techniques to get a more thorough coverage of the input and/or output parameters, for example equivalence partitioning and boundary value analysis for constraints on the parameters.

## 4.3 Defect-Based Techniques

In defect-based testing we are looking at the types of defects we might find in the product under testing. The techniques are therefore starting from previous experience, rather than the expected functionality or the structure of the test product.



The techniques may therefore be less systematic than the previously discussed techniques, since it is usually not possible to make exhaustive collections of expected defects. The determination of defect-related coverage is hence also less comprehensive: Since there is no absolute amount of expected defects, only what we have chosen or selected as expected defects, the coverage is relative to that number.

The defect-based techniques covered here are:

- ▶ Taxonomies
- ▶ (Fault injection and mutation—not part of the ISTQB syllabus)

### 4.3.1 Taxonomies

A taxonomy is an ordered hierarchy of names for something. In this context it is an ordered hierarchy of possible defect types.



Such a taxonomy is a sort of checklist over defects to look for, and it is used to design test cases aimed at finding out if these defects are present in the product under testing.

Many people have worked on defect taxonomies, starting with Beizer's bug taxonomy defined in the late 1980s. This taxonomy is quite comprehensive and lists possible defects in a four-level hierarchy with identification numbers. The first two levels are:



- 1 Requirements
  - 11 Requirements incorrect
  - 12 Requirements logic
  - 13 Requirements, completeness
  - 14 Verifiability
  - 15 Presentation, documentation
  - 16 Requirements changes
- 2 Features and Functionality
  - 21 Feature/function correctness
  - 22 Feature completeness
  - 23 Functional case completeness
  - 24 Domain bugs
  - 25 User messages and diagnostics
  - 26 Exception conditions mishandled
- 3 Structural Bugs
  - 31 Control flow and sequencing
  - 32 Processing
- 4 Data
  - 41 Data definition and structure
  - 42 Data access and handling
- 5 Implementation and Coding
  - 51 Coding and typographical
  - 52 Style and standards violation
  - 53 Documentation
- 6 Integration
  - 61 Internal interfaces
  - 62 External interfaces, timing, throughput
- 7 System and Software Architecture
  - 71 O/S call and use
  - 72 Software architecture
  - 73 Recovery and accountability
  - 74 Performance
  - 75 Incorrect diagnostics, exceptions
  - 76 Partitions, overlay
  - 77 Sysgen, environment
- 8 Test Definition and Execution
  - 81 Test design bugs
  - 82 Test execution bugs
  - 83 Test documentation
  - 84 Test case completeness

This taxonomy covers the entire development life cycle and may also be useful as a checklist for early static test, for example of requirements and design, as well as for static tests of the tests, specification.

Another defect taxonomy is given in IEEE 1044 in the categorization for incidents to be provided during the investigation phase of an incidents life cycle. IEEE 1044 is discussed in detail in Chapter 7. This taxonomy has up to three levels, of which only the first is provided here with the corresponding codes:

IV310	Logical problem
IV320	Computation problem
IV330	Interface/timing problem
IV340	Data-handling problem
IV350	Data problem
IV360	Documentation problem
IV380	Document quality problem
IV390	Enhancement
IV398	Failure caused by fix
IV399	Performance problem
IV400	Interoperability
IV401	Standards conformance
IV402	Other problem



Taxonomy testing is not a terribly effective test technique, especially not if the taxonomy used is a standard one, not taking the nature of the specific development process and product into account.

The taxonomies shown here, and others to be found in the testing literature are, however, very useful as starting points for making your own taxonomy or checklist of possible defects.

The coverage element for taxonomy testing is the listed defects and the coverage is calculated as the percentage of these used for designing test cases.



### 4.3.2 Fault Injection and Mutation

*Note: This technique is not part of the ISTQB syllabus, but included here because I sometimes find it useful.*

Fault (or defect) injection and mutation are a type of technique where the product under test is changed in controlled ways and then tested. This technique type is used to assess the effectiveness of the prepared test cases, rather than actually looking for defects.



Fault injection is also known as fault seeding or bebugging. In this technique defects are deliberately inserted into the source code, either by hand or by the use of tools.



The defects inserted may be inspired by a checklist or a defect taxonomy. The product is then tested, and it is determined how many of the injected defects are found and how many other defects are found. These numbers are used to estimate how many real defects are still left in the product.

**Ex.**

In a set of components 50 defects are injected prior to component testing. The component testing reveals

Injected defects: 26 and New defects: 83

Based on this it is estimated that there remain 77 defects in the components and more tests should be designed, if this is not acceptable.

In mutation testing so-called one-token defects, such as “<” replaced with “<=,” are made in the components. For each of these defects a new version of the affected components is created and tested to see if the prepared test cases reveal the defects. If they don’t they will have to be examined and corrected to find the planted defects—and one hopes, more real defects as well.

The coverage element for this type of testing technique is the injected defects and the coverage is calculated as the percentage of these found.

The drawback of the fault injection and mutation technique type is that the inserted defects are not necessarily realistic. It can also be a fairly big task to define and inject defects compared to the results to be gained.



It must also be noted that even though the technique type helps us identify more defects of the inserted types, there is a number of defects where it is of no use, for example, defects caused by omissions in the code or misunderstandings of the requirements.

Fault injection may also be applied to data, in the sense that data may be edited to be wrong compared to the expected data.

It is absolutely essential when using fault injection and mutation that a good configuration management system is in place. It must be clear what the “correct” code is and what code has been deliberately changed.



## 4.4 Experience-Based Testing Techniques

Faults are sly!

No matter how well we use the test case design techniques we cannot catch all the faults. There are many reasons for this.

One is that not all failures occur every time the same action is performed. Sometimes a failure only occurs when we have performed the same action several times.

**Ex.**

I use home banking when I pay my bills. I enter the details for a bill, hit “OK,” and then the details are presented in a form for my endorsement, and I can go on to the next bill. But, if I have more than eight bills to be paid at the same time, a failure occurs. For the ninth bill the endorsement form is blank! I endorse anyway, however, because the endorsement itself still works.

Another reason why we can miss faults is something called coincidental correctness. We may happen to choose an input, for example, when choosing an input in an equivalence partition that does not reveal a fault.

An illustration of this is a test of the formula “ $n^n$ ” ( $n$  to the power of  $n$ ). Unfortunately the programmer has misunderstood the formula and implemented it as “ $n+n$ .” If we happen to choose the input value of 2, the expected result is 4 ( $2 \times 2 = 4$ )—but alas,  $2 + 2$  also equals 4.



We also need to be aware that identical functionality may or may not be implemented identically. Many are the systems where, for example, date handling has been implemented by different implementations groups for different subsystems.

Furthermore, rare or fringe situations may be overlooked or deliberately left out in the specification of the structured test cases.

The sum of all this is that systematic testing is not enough! Since faults are sly we have to attack them in unpredictable ways.

This is where the nonsystematic testing techniques come in as a valuable supplement to systematic testing techniques. The nonsystematic techniques to be discussed here are:

- ▶ Errorguessing
- ▶ Checklist-based
- ▶ Exploratory testing
- ▶ Attacks



These techniques may be used before the systematic techniques to assess test readiness by uncovering “weak” areas. This can also be used as the input to initial risk analysis. The techniques may also be used after the systematic testing as a final “mopping-up,” hopefully providing extra confidence.

Nonsystematic testing techniques must NEVER be the only technique to be used.



#### 4.4.1 Error Guessing

Error guessing is a test technique where the experience of the tester is used to anticipate what faults might be present in the test object as a result of errors made, and to design tests specifically to expose them.



This means that the tester uses his or her experience gained from the structured tests that have already been executed or from other test assignments to guess where faults may remain in the test object.

The tester has to think creatively—out of the box, over the borders, around the corners—both in relation to how the structured tests have been structured, how the test object has been produced, and the nature of the faults already found.



In relation to the testing approach we could try to find alternative approaches, and ask ourselves:

- ▶ How could this be done differently?
- ▶ What would it be completely unlikely to do?
- ▶ What assumptions may the testers, who performed the structured tests, have made?

In relation to how the test object has been produced, we could ask ourselves:

- ▶ What assumptions may the analysts or the programmers have made?
- ▶ What happens if I use a value of 0 (both input and output)?
- ▶ What about cases of “none” and “one” in lists?
- ▶ What happens if I go over the limit?
- ▶ Are there any cases of “coincidence,” for example, same value twice or same value for all?

The faults we have already found can be exploited to spur new ideas. We can for example ask ourselves:

- ▶ Are there other faults like this?
- ▶ Are there any reverse faults?
- ▶ Are there any perpendicular faults?

All these questions and many more can be assembled and maintained in checklists.

The coverage for error-guessing testing is related to the tester’s experience base and not easily documented.

#### 4.4.2 Checklist-Based

Checklists can hold lists of possible faults that are known to escape the systematic test. They are formed and maintained by experience—lessons learned from previous projects; and they are a valuable asset in an organization. Checklists may be used for designing both static and dynamic tests, and they can be used as a good starting point for for example risk identification or error guessing brain storms.

Special checklists may be defect taxonomies or rules derived from standards, for example, an internal standard for user interfaces.

Coverage for checklist-based testing is related to the contents of the checklists used. The coverage for a specific test may be calculated as the percentage of the items in the list(s) covered by the test.

An example of a checklist is the list shown below for CRUD testing. The



abbreviation CRUD refers to the life cycle of data entities in a product, namely:

- ▶ Create
- ▶ Read
- ▶ Update
- ▶ Delete

CRUD testing, that is testing all the data entities according to their life cycles, is important.

The life cycles for data entities are usually not sufficiently specified in the data requirements. CRUD testing therefore starts with helping the analysts to specify sufficient requirements. The actual CRUD testing is sometimes on the verge of experience-based testing, because it may be based on previous experience of where CRUD faults appear and on related checklists.

The CRUD checklists provided below may be used as inspiration both for defining data requirements and for guiding CRUD testing.



CRUD checklist examples are shown here:

Concerning *creation of data* we may ask:

- ▶ Is it possible to create the first?
- ▶ Are the contents correct?
- ▶ Is it possible to create a new among existing?
- ▶ Are the contents correct?
- ▶ Is it possible to create the last (the highest number)?
- ▶ Are the contents correct?
- ▶ Is it possible to create more than what is allowed?
- ▶ Is it possible to create if you are not allowed to create?



Concerning the *reading of data* we may ask:

- ▶ Is it possible to read the created data?
- ▶ Is it possible to find the data in all the ways it is supposed to be found?
- ▶ Is it impossible to read data if you are not allowed?

Concerning the *updating of data* we may ask:

- ▶ Is it possible to change where it should be possible?
- ▶ Is everything saved?
- ▶ Is the change reflected everywhere?
- ▶ Is it impossible to change in places where it should be impossible?

Concerning the *deletion of data* we may ask:

- ▶ Is it possible to delete where it should be possible?
- ▶ Is everything deleted?
- ▶ Is it possible to delete all that should be deletable?
- ▶ Do all deletes have the correct cascading effects?
- ▶ Is data that should not be deleted properly protected?

#### 4.4.3 Exploratory Testing

Sometimes it is worthwhile to search in a structured way and use the results to decide on the future course as they come in. This is the philosophy when people are looking for oil or mines, and it is the philosophy in exploratory testing.



Exploratory testing is testing where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. In other words exploratory testing is simultaneous:

- ▶ Learning
- ▶ Test design
- ▶ Test execution

Exploratory testing is an important supplement to structured testing. As with all the nonsystematic techniques it may be used before the structured test is completely designed or when the structured test has stopped.

It is important that the course of the exploratory testing is documented, so that we can recall what we have done. Imagine if drillings for oil were not documented, or if the search for mines in a field were not documented—it would be a waste of time and money. The idea in exploratory testing is not that it should not be documented, but that it should be documented as we go along.



*Exploratory testing is not for kids!*—nor for inexperienced testers.

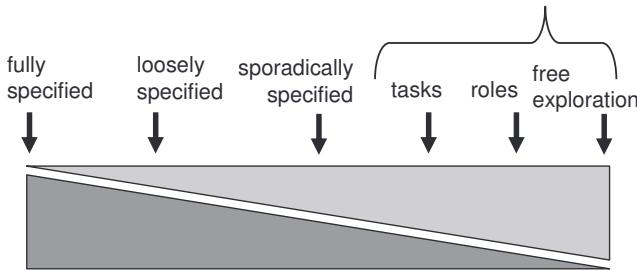
Extensive testing experience and knowledge of testing techniques and typical failures are indispensable for the performance of an effective exploratory test. It is also an advantage if the tester has some domain knowledge.

The exploratory tester needs to be able to analyze, reason, and make decisions on the fly; and at the same time have a systematic approach and be creative. The tester also needs some degree of independence in relation to the manufacturing of the system—the programmer of a system cannot perform exploratory testing on his or her “own” system.

Perhaps most importantly the exploratory tester must have an inclination towards destruction. Exploratory testing will not work if the tester is “afraid” of getting the system to fail.

#### 4.4.3.1 Degrees of Exploratory Testing

One of the forerunners in exploratory testing is the American test guru James Bach. He has defined various degrees of exploration as illustrated in the figure below.



Furthest to the right, we have the totally free exploration. The tester simply sits before the system and starts wherever he or she feels like.

A step to the left, we find the exploratory testing guided by roles. Here the tester attacks the system under testing assuming a specific user role. This could, for example, be the role of an accountant, a nurse, a secretary, an executive manager, or any other role defined for the system. This provides a starting point and a viewpoint for the testing, which is exploratory within the framework of the role.

Even further to the left is the exploratory testing guided by a specific task. Here the tester narrows the framework for the testing even more by testing within the viewpoint of a specific task defined for a specific role for the system under testing.

On the borderline between exploratory testing and structured testing we have the sporadically specified test. Here the tester has sketched the test beforehand and takes this as the starting point and guideline for the performance of the exploratory testing.

#### 4.4.3.2 Performing Exploratory Testing

No matter which degree of exploration we use, we have to follow the principles in the general test process. We must plan and monitor; we must specify, execute, and record; and we must check for completion.

In the planning we consider what we are going to do and who is going to do it. We must choose the degree of exploration and describe the appropriate activities. The testing activities should be divided into one-hour sessions. If the sessions are shorter we risk not getting an effective flow in the exploration; if they are longer we get tired and the effectiveness goes down.

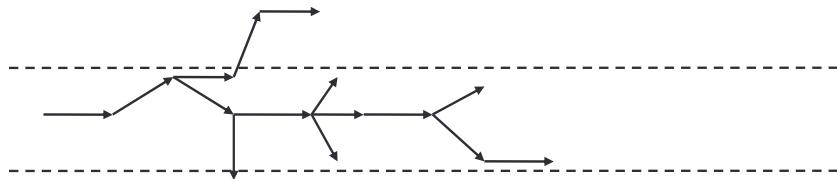
It is important to make sure that the tester or testers are protected during the sessions. There should be no phones or other interruptions to disturb the



flow of the testing.

The test specification, execution, and recording are done simultaneously during the exploratory testing session. Within the given degree of exploration the tester should allow him- or herself to get distracted—you never know what you may find.

The course of the testing session may be illustrated like this:



Stock must be taken from time to time to verify that we are on track.

For each session we must:

- ▶ Take extensive notes and attach data files, screen dumps, and/or other documentation as appropriate
- ▶ Produce an overview over findings
- ▶ Reprioritize the remaining activities

The exploratory testing can stop when we have fulfilled our purpose.

#### 4.4.3.3 Exploratory Testing Hints

There are a few weaknesses in exploratory testing, of which we need to be aware. These weaknesses are part of the reasons why exploratory testing must be a supplement only to structured and specified testing.



*The weaknesses in exploratory testing are:*

- ▶ Exploratory testing does not support automated test, and hence regression testing, very well.
- ▶ Because the exploratory testing is not specified in advance it cannot provide feedback to design before the design is actually implemented.
- ▶ Even when accompanied by extensive notes there is usually no firm documentation of test coverage for the exploratory testing.
- ▶ Exploratory testing can be very difficult to use for complex functionality; the main thread may easily be lost.
- ▶ Note taking is very difficult when working interactively.

The last weakness may be overcome by performing exploratory testing as *pair testing*. This can work really well and it has a number of benefits. Pair work sparks more ideas as the two testers inspire each other, and it enables mutual learning—even when the testers have different levels of experiences.



Extroverted people get more energy from working together, and others are less likely to interrupt a pair.

The biggest benefit is perhaps that two testers can be focused on two tasks at the time: One can follow an idea, and the other can take notes. The focuses should switch between the two testers at regular intervals.



There are of course also a few *disadvantages to pair testing*. The main one is, that a divided responsibility is no responsibility—if two people share a responsibility they carry 2% each.

As some people are extroverted, some people are introverted. Introverted people get drained for energy by working closely together with others.

If the difference in experience is too large it may cause “abandonment” by the lesser experienced, because he or she may simply opt out. On the other hand different opinions held by equally “strong” testers may block progress.



#### 4.4.4 Attacks

The attacks technique is a form of security testing, testing how resistant a product is to those who want to break into it in various ways and for various reasons, ranging from incidental mistakes, over “fun,” to serious crime.

Security testing is getting more and more powerful and sophisticated. As the market for e-commerce and e-business is growing and more and more other applications get Web access, the need for secure systems is growing. Even so, we are still constantly at least one step behind, and checklists of attacks are very valuable both to analysts defining requirements and to testers.

A product is vulnerable at the places where there is an opening into it; that is where the product has interfaces. The interfaces a product can have include, but are not limited to:

- ▶ User interface
- ▶ Operating system
- ▶ API (application programming interface)
- ▶ Data storage for example file system

Attacks are used to find areas where the product will fail due to misuse or defects in these interfaces.



James Whittaker is one of the pioneers of attacks, and he has created long lists of useful attacks. A few examples are listed here, grouped by type:

User interface attacks:

- ▶ Apply inputs that force all the error messages to occur
- ▶ Apply inputs that force the software to establish default values
- ▶ Explore allowable character sets and data types
- ▶ Overflow input buffers

Stored data attacks:

- ▶ Apply inputs using a variety of initial conditions
- ▶ Force a data structure to store too many or too few values
- ▶ Investigate alternate ways to modify internal data constraints

Media based attacks:

- ▶ Fill the file system to its capacity
- ▶ Force the media to be busy or unavailable
- ▶ Damage the media

Other ways of attacking a product may be trying to make unauthorized

- ▶ Access to and control over resources, such as restricted files and data
- ▶ Execution of programs or transactions
- ▶ Access to and control over user accounts
- ▶ Access to and control over privilege management
- ▶ Access to and control over network management facilities



The advantage of using attack-driven testing is that security holes can be closed before they are found by attackers. A pitfall is that this may create a false sense of security. There is no end to the imagination of those who want to do wrong, and we have to stay constantly alert to new weak spots. This is why checklists of any kind must be kept up-to-date with new experiences.



There is more about security testing in Sections 5.1.4 and 5.2.2.



## 4.5 Static Analysis

Static analysis is a testing type where, in contrast to dynamic testing, the code under static analysis is NOT executed.

Static testing, especially of code, is usually performed using tool(s), but the only thing being executed during static analysis is the tool. Static test tools are discussed in Section 9.3.4.

Traditionally static testing has been performed on code, but here it is expanded to architecture as well.

### 4.5.1 Static Analysis of Code

Many tools dedicated to static analysis are available on the market or as open source systems. Their capabilities vary a lot and depend very much on the coding language. Some standard development tools, such as compilers or linkers, are able to perform limited static analysis.

The static analysis techniques for code discussed here are:

- ▶ Control flow analysis
- ▶ Data flow analysis
- ▶ Compliance to standards
- ▶ Calculation of code metrics



#### 4.5.1.1 Control Flow Analysis

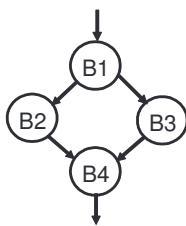
A control flow is an abstract representation of all possible sequence of events (paths) in the execution through a component or a system. The control flow is the basis for many of the structure-based case design techniques described earlier.



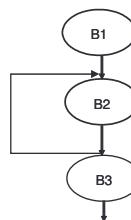
Control flow goes through basic blocs or nodes of code being executed as an entity, with an entry point in the beginning and only there, and with an exit point in the end and only there. The control goes from the starting point and is transferred from one block to another to the end.

It can be very useful to draw a control flow graph from the code (or the requirements). It gives an overview over the decisions points, branches, and paths in a piece of code, and its inherent complexity.

Two simple extracts of control flow graphs are shown here,



More basic blocks in parallel



More basic blocks in parallel

Static analysis tools may be used to draw control flow graphs of larger pieces of code. It is, however, a good idea to train drawing these, because they give a good understanding of how the code is structured.

A control flow graph does not necessarily give an idea of what the code is actually doing, but that is not important anyway. We as testers should concentrate on the structure, not the functionality.

Static analysis tools can find faults in the control flow, typically:

- ▶ “Dead” code (i.e., code that cannot be reached during execution)
- ▶ Uncalled functions and procedures

Both dead code and uncalled functions are quite often found in legacy systems. Undocumented changes and corrections have caused some code to be circumvented but not removed. The cause of the change is since forgotten, but nobody maintaining the code has had the courage (or initiative) to get the unused code removed.

Such unused areas do not present direct risks. They do however disturb maintenance, and they may unintentionally be invoked with unknown consequences.

#### 4.5.1.2 Data Flow Analysis

Data flows through the code; that is what IT is all about. The normal life cycle for a data item, a variable, consists of the phases:



- ▶ Declaration—Space is reserved in memory for the variable value
- ▶ Definition—A value is assigned to the variable
- ▶ Use—The value of the variable is used
- ▶ Destruction—The memory set aside for the value of the variable is freed for others to use

Some static analysis tools can find anomalies in the data flow in relation to the variable life cycle.

Anomalies may, for example, be:

- ▶ Use before declaration
- ▶ Use before definition
- ▶ Redefinition before use
- ▶ Use after destruction

Any of these anomalies should set the alarm clocks ringing, as it may be a sign of something being wrong.

**Ex.**

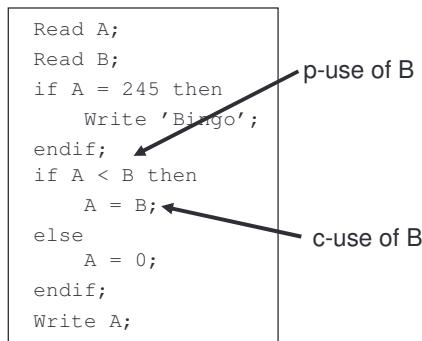
If a variable is defined and then redefined before use, is it because the first or the second definition is superfluous, or is it because a usage statement is missing?

Some programming languages permit variables to be used before they are declared; in this case the first usage will be treated as an implicit declaration.

The phases “definition” and “use” may be repeated many times during the life of a variable.

Two different kinds of usage are defined in data flow analysis:

- ▶ Computation data use = c-use = data not used in a condition
- ▶ Predicate data use = p-use = a data use associated with the decision outcome of the predicate portion of a decision statement (predicate = condition = evaluates to T or F)



A subpath in the flow is defined to go from a point where a variable is defined, to a point where it is referenced, that is, where it is used—whatever kind of usage it is. Such a subpath is called a definition-use pair (du-pair). The pair is made up of a definition of a variable and a use of the variable.

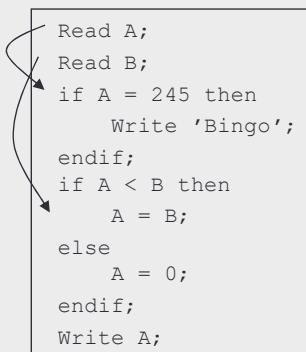


Because there are two kinds of usages, there are three types of du-pairs, namely:

- ▶ Definition → c-use
- ▶ Definition → p-use
- ▶ Definition → use (either c or p)

This example shows two du-pairs:

- ▶ One definition to p use
- ▶ One definition to c use



Data flow testing is testing in which test cases are designed based on variable usage within the code, that is, testing of du-pairs.

The coverage items for this test case design technique are the control flow sub paths and full paths through the code. The following table gives an overview of the types of coverage defined for data flow testing.

	Name	Definition
d	All-definition coverage	percentage of covered subpaths from each variable definition to some use of that definition
duc	All data definition c-use coverage	percentage of covered subpaths from each variable definition to every c-use of that definition
dup	All data definition p-use coverage	percentage of covered subpaths from each variable definition to every p-use of that definition
u	All use coverage	percentage of covered subpaths from each variable definition to every use of that definition (despite the type)
du-path	All definition use path coverage	percentage of covered “simple subpath” from each variable definition to every use of that definition

To design test cases with the data flow test technique we concentrate on one component and for that component we must:

- ▶ Number the lines, if they are not numbered already
- ▶ List the variables
- ▶ List each occurrence of each variable and assign a category (definition, p-use, or c-use)
- ▶ Identify the du-pairs and their type (c-use or p-use)
- ▶ Identify all the subpaths that satisfy the pair
- ▶ Derive test cases that satisfy the subpaths

When the final test procedures are designed from the test cases, they will of course follow a path in the control flow through the code.



There are a few pitfalls that we need to be aware of in connection with data flow analysis. Data items (variables) may be composed of a number of single variables. This is, for example, the case for arrays, which are ordered sequences of individual data items. If we ignore the constituents of composite variables like arrays and treat them as one data item, we greatly reduce the effectiveness of data flow testing. Tools will typically respond to an array or record it as a single data item rather than as a composite item with many constituents.

An important problem concerning data flow analysis is that sometimes static analysis tools will report a large number of anomalies that are not in

fact caused by anything being wrong. These “false alarms” may hide the real problems in the sheer number of alarms.

#### 4.5.1.3 Compliance to Coding Standard

A coding standard is a guideline for the layout of the code being written in an organization.

Most static analysis tools can check for standard coding style violations, for example, missing indentation in IF statements. Some of the more sophisticated tools allow the definition of specific coding standards to check for.

It may seem trivial that coding standards should be defined and adhered to, but experience shows that it has several advantages, such as:

- ▶ Fewer faults in the code, because a good layout of the code enables the programmer to keep an overview of what he or she is writing
- ▶ Easier component testing, because a well-structured code is easier to define test cases for when using white-box techniques
- ▶ Easier maintenance, because it is faster for others to overtake code if all code is written in an identical style

There are no disadvantages in requiring adherence to a coding standard, other than the programmers having to get used to it. It is even possible to get tools to format “rough” code according to coding standards.

#### 4.5.1.4 Calculation of Code Metrics

Static analysis tools can provide measurements of different aspects of the code like:

- ▶ Size measures—Number of lines of code, number of comments
- ▶ Complexity
- ▶ Number of nested levels
- ▶ Number of function calls—Fan-out

Lines of code (LOC) can be reported as can lines of document lines. The ratio between these may be calculated as a derived measurement.

Measurements related to code lines may be difficult to work with. The sheer definition of a “code line” is not always as simple as it may sound. Numbers of code lines may also easily be manipulated by the programmers, if the measurements are used to quantify their efficiency.

*Code complexity* is a measure for how “tangled” the code is. The complexity is related to the number and types of decisions in the code. It is interesting from a testing point of view because it has an impact on the test effort: the higher the complexity, the more test cases to get high decision and condition coverages.



The most commonly used complexity measure is McCabe's Cyclomatic Complexity. It was introduced by Thomas McCabe in 1976 and is often simply referred to as complexity, as CC or as McCabe's complexity.

McCabe's Cyclomatic Complexity is a measure - a single ordinal number—of soundness of components. It measures the number of linearly independent paths through the code. It is intended to be independent of language and language formats.



The original definition of McCabe's Cyclomatic Complexity is:

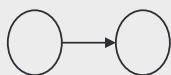
$$MCC = (L - N) + 2 * B$$

where

*L*: Number of branches (lines in the flow graph)

*N*: Number of sequential blocks (basis blocks or nodes)

*B*: Number of broken sequences (this is used when we measure for more than one component. This is very rarely done, so *B* is usually = 1)



In this simple flow graph we have

$L = 1$     $N = 2$    and    $B = 1$    and therefore

$$MCC = (1 - 2) + 2 \times 1 = 1$$



For this flow graph



we have  $L = 4$     $N = 4$    and    $B = 1$ , and therefore

$$MCC = (4 - 4) + 2 \times 1 = 2$$

There are more simple ways of calculating McCabe's Cyclomatic Complexity:

One way is based on a count of decisions and decision outcomes. It could be expressed like this

$$MCC = (\text{SUM}(\text{branches} - 1) \text{ for all decisions}) + 1$$



IF statements always have two branches, so if we only have IF statements in the code we are looking at, we'll get

$$MCC = \text{SUM}(1 \text{ for all IF statements}) + 1 = \text{number of IF statements} + 1.$$

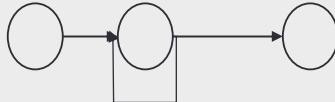
Constructions like CASE have more branches, so if we have a piece of code with two IF statements and 1 CASE statement with 10 possible outcomes, we would have  $MCC = 1 + 1 + 9 + 1$ .

The third way of calculating McCabe's Cyclomatic Complexity is by counting so-called regions in the code. A region is a closed area found in the flow graph.

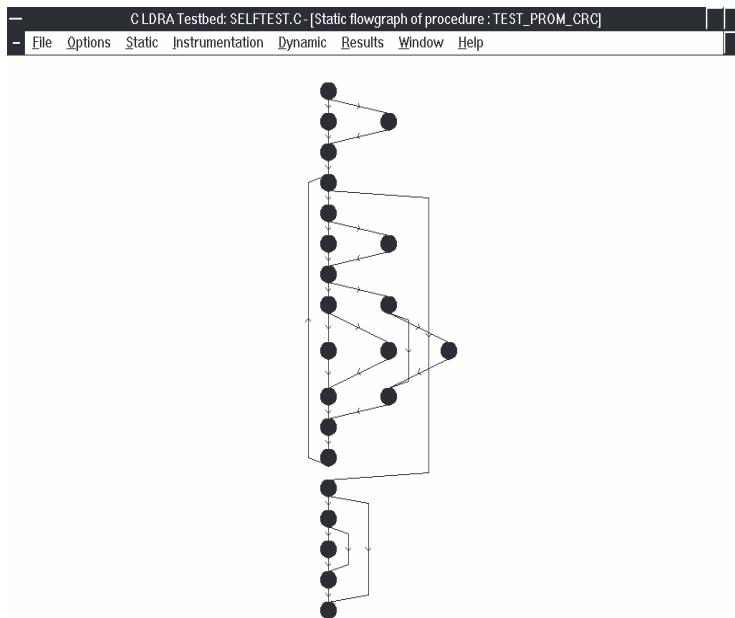
The expression is

$$\text{MCC} = \text{number of regions} + 1$$

In the flow graph shown here, we have 1 region and we therefore have MCC = 2.



The graph shown below is created by a static analysis tool. The tool has calculated the McCabe's CC for the component to be 9.



McCabe's Cyclomatic Complexity is a good indicator for the test effort. Mathematical analysis has shown that it gives the exact number of tests needed to test every decision point in a program for each outcome.

Some testers have experienced that there is a positive linear connection between the number of faults and the McCabe's CC (i.e., that the higher the CC the higher the number of faults). Others have found that the connection is rather that a low CC and a high CC indicate relatively fewer faults, while more faults are found in the areas where the CC is around average. Try to get your own measurements for this!



A rule of thumb says that the McCabe's CC should not be more than 10. If we find a higher CC in a component it should be revisited—maybe it could be restructured or divided. McCabe's CC measures have however been seen up to more than 3,000, so we should also use visual judgment of the control flow graphs to determine the test effort.

Cyclomatic complexity can be used for other purposes than test planning, for example, in risk analysis related to:

- ▶ Code development
- ▶ Implementation of changes in maintenance
- ▶ Reengineering of existing systems

Other complexity measures than McCabe's exist. They are, however, rarely used, except perhaps for the Halstead complexity measure, an algorithmic complexity measure.

#### 4.5.2 Static Analysis of Architecture



In the recent testing literature the static analysis objects have been expanded to include the architecture of the product as well as the classic code object.

The code has a structure that can be analyzed, and so has the product or system as a whole. This may, for example, be:

- ▶ Structure of components in the product calling/using each other
- ▶ Menu structure of a graphical user interface for a product
- ▶ Structure of pages and other features in a Web product



The first and the second of these types may be tested with the help of a tool. We can therefore say that static analysis may be used to test these aspects of products.

##### 4.5.2.1 Static Analysis of a Web Site



A Web site is a hierarchical structure composed of Web pages with elements such as tables, text, pictures, and links to other pages, both internally to the Web site itself and to external Web sites. The structure of a Web site is described in HTML: Hyper Text Mark-up Language.

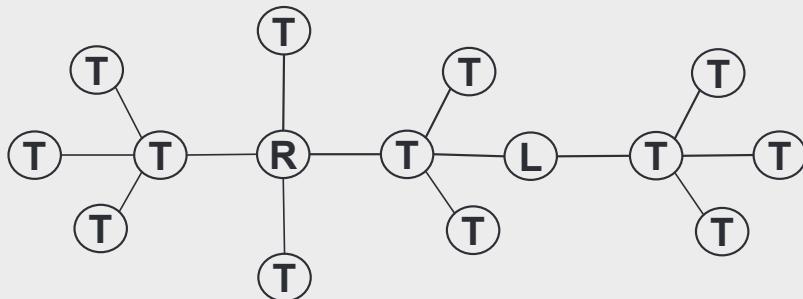
Many tools exist to analyze the HTML code and the structure and detailed composition of Web sites, that is to perform static analysis of Web sites.

The stakeholders for the static analysis information are both analysts, designers, testers, and those in charge of the maintenance of the Web sites, the Web masters.

One of the results of such an analysis is a graph showing the structure of the Web site. A graph can provide an overview of the tree structure or hierarchy of the site and show the depth, complexity, and balance of the structure.

A very simple Web site is shown here. The R is the root, a T illustrates a non-link tag, for example a table, and an L illustrates a link.

**Ex.**



The graphs can be very colorful and look like the most amazing fireworks. The information extracted from the graphs can be used to:

- ▶ Test whether requirements for the Web site, for example depth and balance of the Web site, have been fulfilled, or if the Web site needs to be restructured
- ▶ Estimate test effort
- ▶ Assess usability
- ▶ Assess maintainability

A rule of thumb is that the more balanced the Web site tree is, the lower the test and maintenance efforts are going to be, and the higher the usability and vice versa.

Web sites are extremely dynamic products. The static analysis of a site prior to its release is only the tip of the iceberg of testing and monitoring of a Web site. Dynamic analysis of the behavior of a site should be performed on a regular basis to identify problems like:



- ▶ Broken links
- ▶ Incomplete downloads
- ▶ Deteriorating performance
- ▶ Orphaned files

Furthermore, information may be obtained about:

- ▶ Access patterns (where do users start and where do they end their visits)
- ▶ Navigation patterns
- ▶ Activity over time
- ▶ General performance
- ▶ Page-loading speed



The Web site structure should also be monitored at regular basis, as Web sites are likely to grow and change structure in an ad hoc manner.

#### 4.5.2.2 Call Graphs

The analysis of the architectural structure of a product is on the borderline between design and testing. In the architectural design the product is decomposed into smaller and smaller components. Components call other sub routines or functions in other components; in object-oriented design we say that classes provide methods for others to use. These dependencies or couplings can be illustrated in call graphs produced by tools.

Static architecture analysis tools can also provide measurements related to the coupling, such as:



- ▶ Fan-in: Number of calls or usages of a specific function or methods made from other functions or methods (or the main program)
- ▶ Fan-out: Number of calls or usages of functions or methods made from a specific component
- ▶ Henry and Kafura metrics: Coupling between modules
- ▶ Bowles metrics: Module and system complexity
- ▶ Troy and Zweben metrics: Modularity or coupling; complexity of structure
- ▶ Ligier metrics: Modularity of the structure chart

These metrics may be more detailed and specific depending on the architectural paradigm.

Both the call graphs and the corresponding measurements provide valuable information to the test planning. This information may for example be used to:

- ▶ Determine integration high-risk areas (high fan-out and fan-in)
- ▶ Determine detailed integration testing sequence
- ▶ Determine intercomponent testing approach

Especially in object-oriented architecture, it is relevant to know the dy-

namic dependencies, that is, how often a method is used during execution of the product. This information can be combined with the static architectural information to strengthen the test planning even more.

## 4.6 Dynamic Analysis

Dynamic analysis is the process of evaluating a system or a component based upon its behavior during execution.

There are two aspects of dynamic analysis, namely:

- ▶ Dynamic—Code is being executed
- ▶ Analysis—Finding out about the nature of the object and its behavior

Dynamic analysis cannot be done without tool support. Chapter 9 discusses the testing tool types.

A dynamic analysis tool instruments the code in order to catch the relevant run-time information. This means that extra code is added to the code written by the developer. The effect of this is that it is not strictly the “real” code we are analyzing. In most cases this is without any importance. The instrumentation may, however, have an adverse impact on performance, and that can pose problems if we are testing time-sensitive real-time software.

In dynamic analysis we prepare the software code we are going to analyze. The component or larger object is then executed. This execution may be execution of test cases for other test purposes, or it may be execution of specific scripts produced for the analysis.

The great advantages of dynamic analysis are that it

- ▶ Provides run-time information otherwise difficult to obtain
- ▶ Provides information as a by-product of dynamic testing
- ▶ Finds faults that it is almost impossible to find in other ways

The dynamic analysis tool reports on what is going on during execution: It provides run-time information about the behavior and state of software while it is being executed. The information we can get from this covers:

- ▶ Memory handling and memory leaks
- ▶ Pointer handling
- ▶ Coverage analysis
- ▶ Performance analysis

### 4.6.1 Memory Handling and Memory Leaks

Memory handling is concerned with allocation, usage, and deallocation of memory.

The tools can detect memory leaks, where memory is gradually being



filled up during extended use. This happens if we keep on allocating memory in the program and forget to deallocate memory when we no longer need it. If a program with a memory leak keeps on running we can end up with no more available memory. This will cause a failure.

Memory is automatically deallocated when the program execution stops; this is one of the reasons why memory leaks are not always detected during dynamic testing. When we test we rarely run the program for longer periods, as it might be run in real life.

The advantage of dynamic analysis in connection with memory leaks is that the analysis can detect the possibility of memory leaks long before they actually happen.

### 4.6.2 Pointer Handling

Pointers are used to handle dynamic allocation of memory. Instead of working with a fixed name or address of a variable we let the system find out where the actual location is, and we use a pointer to point to this location in the program.

The usage of pointers can go wrong in a number of ways. Pointers can, for example, be unassigned when used, lose their object, or point to a place in memory to which it is not supposed to point, for example beyond an array border or into some protected part of memory. A wrong pointer can cause for example part of the program or data stored in memory to be overwritten.

It may be very difficult to find the defects, underlying failures caused by wrong pointer handling. These failures have a tendency to be periodic, that is, the program may run for a long time and then suddenly start to give wrong results or even crash.

Dynamic analysis tools can identify unassigned pointers, and they can also detect faults in pointer arithmetic, for example, if the addition of two pointers results in a pointer that points to an invalid place in memory.

### 4.6.3 Coverage Analysis

The coverage obtained in an executed test can be measured by analysis tools. These tools provide objective measurement for some structural or white-box test coverage metrics, for example:

- ▶ Statement coverage
- ▶ Branch coverage

The tools provide objective measurements to be used in the checking against test completion criteria in a fast and reliable way.

Some tools can also deliver reports about uncovered areas. The more fancy ones produce colored reports where covered code is shown in one color and uncovered code in another. This is a great help when more test cases must be designed to obtain a higher coverage.

#### 4.6.4 Performance Analysis

All too many products have no or insufficient performance requirements and turn out to be unable to cope with real-life volumes and loads. Performance analysis aims at measuring the performance of a product under the controlled circumstances before the product is released.

It is much better to get these aspects tested before the product breaks down when the first set of users starts using it. Tools may be used to measure what the performance is under given circumstances.

The performance testing tools can provide very useful reports based on collected information, often in graphical form. The tools can provide information about “bottle neck” areas relatively inexpensively before the product hits the real world.

Some companies have specialized in performance testing and will test products using these tools. This is a very useful alternative to investing in the tools yourself.



### 4.7 Choosing Testing Techniques

The big question remaining after all these descriptions of wonderful test case design techniques is: Which testing technique(s) should we use? The answer to that is: *It depends!*



There is no established consensus on which technique is the most effective. The choice depends on the circumstances, including the testers' experience and the nature of the object under testing.

With regard to the testers' experience it is evident that a test case design technique that we as testers know well and have used many times on similar occasions is a good choice. All things being equal there is no need to throw old techniques overboard. Despite the general feeling that everything is changing fast, techniques do not usually change overnight. On the other hand, we need to be aware of new research and new techniques, both in development and testing becoming available from time to time.

A little more external to the testers' direct choice is the choice guided by risk analysis. Certain techniques are sufficient for low-risk products, whereas other techniques should be used for products or areas with a higher risk exposure. This is especially the case when we are selecting between structural or white-box techniques. Testing and risk are discussed in Section 3.5.



Even further away from the testers, the choice of test techniques may be dictated by customer requirements, typically formulated in the contract. There is a tendency for these constraints to be included in the contract for high-risk products. It may also be the case for development projects contracted between organizations with a higher level of maturity. In the case of test case techniques being stipulated in a contract, the test responsible should have

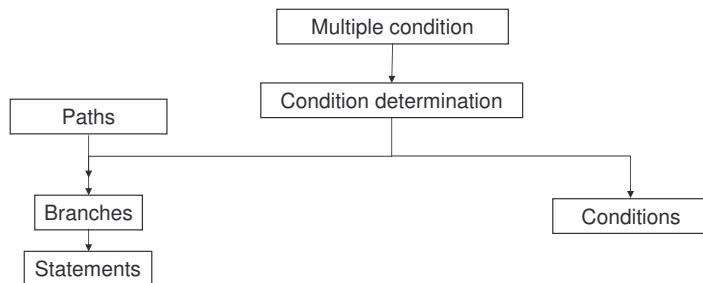
had the possibility of suggesting and accepting the choices.

Finally the choice of test case design techniques can be guided or even dictated by applicable regulatory standards.

#### 4.7.1 Subsumes Ordering of Techniques

It is possible to define a sort of hierarchy of the structural test case design techniques based on the thoroughness of the techniques at 100% coverage.

This hierarchy is called the subsumes ordering of the techniques. The verb “subsume” means “to include in a larger class.” The subsumes ordering show which techniques are included in techniques placed higher up in the order. The ordering is shown here.



The ordering can only be read downwards. We can for example see that condition determination subsumes branches. Paths also subsume branches; but we cannot say anything about the ordering of condition determination in relation to paths.

The subsumes ordering does not tell us which technique to use, but it shows the techniques’ relative thoroughness. It also shows that it does not make sense to require both a 100% branch and a 100% statement coverage, because the latter will be superfluous.

#### 4.7.2 Advice on Choosing Testing Techniques

No firm research conclusions exist about the rank in effectiveness of the functional or black-box techniques. Most of the research that has been performed is very academic and not terribly useful in “the real testing world.”

One conclusion that seems to have been reached is: There is no “best” technique. The “best” depends on the nature of the product.

We do, however, know with certainty that the usage of *some technique is better than none*, and that a combination of techniques is better than just one technique.

We also know that the use of techniques supports systematic and meticulous work and that techniques are good for finding possible failures. Using test case design techniques means that they may be repeated by others with



approximately the same result, and that we are able to explain our test cases.

There is no excuse for not using some techniques.

In his book *The Art of Software Testing*, Glenford J. Meyers provides a strategy for applying techniques. He writes:

- ▶ If the specification contains combinations of input conditions, start with cause-effect graphing
- ▶ Always use boundary value analysis (input and output)
- ▶ Supply with valid and invalid equivalence classes (both for input and output)
- ▶ Round up using error guessing
- ▶ Add sufficient test cases using white-box techniques if completion criteria has not yet been reached (providing it is possible)



As mentioned earlier some research is being made into the effectiveness of different test techniques.

Stuart Reid has made a study on the techniques equivalence partitioning, boundary value analysis, and random testing on real avionics code. Based on all input for the techniques he concludes that BVA is most effective with 79% effectiveness, whereas EP only reached 33% effectiveness. Stuart Reid also concludes that some faults are difficult to find even with these techniques.

## Questions

1. Why is it a good idea to use test techniques?
2. What is the other, perhaps more common, name for specification-based testing techniques?
3. What is the basic idea in equivalence partitioning?
4. Why may equivalence partitioning reduce the number of test cases?
5. What is boundary value analysis?
6. What are the coverage elements for equivalence partitioning and boundary value analysis?
7. What should happen to invalid values?
8. Where do we find most faults in connection with equivalence partitioning and boundary value analysis?
9. When is domain analysis used?
10. What are the four points defined in domain analysis?
11. What are the coverage elements for domain analysis?
12. How many columns does a decision table have depending on the number of input conditions?
13. What is the coverage element for decision tables?
14. How can you fill in a decision table?
15. What is one of the main application areas for decision tables?

16. What is a cause-effect graph?
17. How is cause-effect graph coverage defined?
18. What are the building blocks of a cause-effect graph?
19. What does an arch mean in a cause-effect graph?
20. What does a transition consist of?
21. What is the coverage measure for state transition testing?
22. What is the lowest coverage level in state transition testing?
23. What should be tested on top of the cases dictated by the coverage?
24. What is the main difficulty with state machines?
25. What are the two types of nodes in a classification tree?
26. What are the two rules that must be observed when a classification tree is constructed?
27. With what does a classification tree end?
28. What is the coverage element for a classification tree?
29. What should be considered input in a classification tree?
30. When can pairwise testing be used?
31. What is the coverage element in pairwise testing?
32. What characterizes an orthogonal array?
33. How can an orthogonal array be described?
34. What is the Allpairs algorithm?
35. What is the weakness of pairwise testing?
36. What is a use case?
37. How can a use case be the basis for testing?
38. What is syntax testing used for?
39. Who have defined a notation form for syntax?
40. How can you define invalid syntaxes?
41. What is structure-based testing also known as?
42. How should structure-based testing be used in relation to specification-based?
43. From where must the expected results be derived in structural testing?
44. How does a basic block end?
45. What is a statement?
46. How is statement coverage defined?
47. What is decision testing?
48. How is decision coverage defined?
49. What is a condition?
50. How is condition coverage defined?
51. How many test cases do we usually need to get 100% condition coverage for one decision?
52. How many test cases do we need to get 100% branch condition combination coverage for one decision?

53. What is an optimized expression?
54. What is done when designing test cases for modified condition decision testing?
55. What is a loop?
56. What is a LCSAJ?
57. How do we find the LCSAJs in a component?
58. What are the main pitfalls when working with loops?
59. What is path testing?
60. What should, for example, be tested in path testing?
61. What causes path testing to be very difficult?
62. At which test level(s) is intercomponent testing used?
63. What is the measure for the number of calls made to a specific function?
64. What is a taxonomy?
65. Where can examples of defect taxonomies be found?
66. What can fault injection be used for?
67. What is important in connection with fault injection and mutation testing?
68. What is the basis for experience-based testing?
69. What do we especially need to do in error guessing?
70. What characterizes exploratory testing?
71. What are the three degrees of exploratory testing?
72. What is important during exploratory testing?
73. What are the weaknesses of exploratory testing?
74. How can exploratory testing (maybe) be enhanced?
75. What are we looking for in attack testing?
76. What is static analysis?
77. What is a control flow graph?
78. What can control flow analysis find?
79. What is the normal life cycle for a variable?
80. What is p-use and c-use?
81. How can coverage for data flow testing be measured?
82. What may be a problem with data flow testing?
83. What are complexity measures used for?
84. What are the advantages of coding standards?
85. What are the name and the definition of the most used complexity measure?
86. When should we look at the code as perhaps being too complex?
87. How can we get an overview of a Web site?
88. How is the test effort connected to the “shape” of a Web site?
89. When can we stop monitoring Web sites?
90. What is fan-in and fan-out?

91. What is dynamic analysis?
92. What is a memory leak?
93. What may go wrong with pointers?
94. What is coverage?
95. How can a performance problem be reported?
96. Which test techniques is the best?
97. What is the subsumes order of test techniques?
98. Which test technique is better than none?
99. Which technique does G.J. Meyers recommend as the first one to use?
100. How is the effectiveness of a test after, say, three months in production calculated?

## Appendix 4A Classification Tree Example

Test design item no.: 56		Traces: Req. (1) – (6)						Test cases								
Domain 1	Aspect 1	Domain 2	Aspect 2	Domain 3				tag	1	2	3	4	5	6	7	8
all inputs and types of lists																
input type	pure text							56-1	x							
	pure number							56-2	x	x						
	mixture							56-3	x							
	empty							56-4	x							
match?	no							56-5	x	x	x	x				
	yes	no. of matches	1					56-6	x		x					
			> 1					56-7	x	x						
				type of match	name	how much	all	56-8	x							
					phone no.		some	56-9	x							
								56-10	x	x						
								56-11								
								56-12	x	x	x	x	x			
								56-13					x			
								56-14	x	x	x	x				
								56-15			x					
								56-16	x	x	x	x	x			
								56-17			x					



## Testing of Software Characteristics

A standard definition of the quality of a product is the degree to which the product fulfills the expectations the customers and users have for it. We test to get information about the quality, that is, information about the fulfillment of the specified requirements, expectations, and/or implied needs.

The better the requirements, expectations, and implied needs are known, understood, and documented, and the better these specifications are, the easier it is for the developers to produce a satisfactory product and the easier it is for testers to test it. This applies both for sequential, iterative, and agile development.

This chapter is about quality characteristics or quality attributes, as they are also called. Quality attributes represent a way of structuring and expressing the expectations for a product.

*In a way this chapter is superfluous for testers.* It is, however, a very good idea for testers to understand what quality attributes are and how they may be expressed. With this understanding testers can contribute to the quality of a product in a number of ways from the very start of the development life cycle, for example, by expressing requirements in a testable way, reviewing requirements specifications for completeness and testability, preparing of tests to cover requirements, expectations, and implied needs, and dynamic testing of these.

The quality of a product should be measured both with regard to what the product shall do—the functional quality attributes—and with regard to how the functionality shall present itself and behave—the nonfunctional or functionality-sustaining attributes.

### Contents

5.1 Quality Attributes  
for Test Analysts

5.2 Quality Attributes  
for Technical Test  
Analysts

The ISO 9126 standard, a standard providing a quality model for product quality, lists the following quality attributes (also sometimes called quality factors):

- ▶ Functionality
- ▶ Reliability
- ▶ Usability
- ▶ Efficiency
- ▶ Maintainability
- ▶ Portability

This standard is used as the basis for this chapter.

The test analyst is concerned with the functional and the usability quality attributes, and the technical test analyst is concerned with the other four quality attributes. Both are concerned with security testing, though from different perspectives.



ISO 9126 expresses that compliance to relevant standards and regulations should be verified for all the quality attributes. This is regarded as a part of the requirements specification to be tested under all circumstances and will not be discussed further here.

## 5.1 Quality Attributes for Test Analysts

The functionality is what the product can do. Without functionality we don't have a product at all. The functionality supports the users in their daily work or their leisure, as the case might be.



Functionality may be tested at almost all test levels. In component testing we can test the functionality implemented in single components; in system testing we can test functionality implemented in single systems across a number of integrated components; and in product testing we can test functionality implemented in the entire product. In testing how the functionality meets the expectations we can use most of the testing techniques discussed in Chapter 4.

In ISO 9126 language, the quality attributes cover the existence of a set of functions and their specified properties in the product. The functions are those that satisfy stated or implied needs, from the point of view of a stated or implied set of users.

ISO 9126 breaks the functionality attribute into the following sub-attributes:



- ▶ Suitability
- ▶ Accuracy
- ▶ Interoperability
- ▶ Security

This list may be used as a checklist for producing requirements specifications, as well as for structuring the functionality testing, if this is not being based strictly on a specification. Each of the subattributes is discussed in more detail later.

Usability, a nonfunctional attribute, is handled at the end of the section.

### 5.1.1 Functional Testing

#### 5.1.1.1 Suitability Testing

In suitability testing we test the requirements or needs concerned with the presence and appropriateness of a set of functions for specified tasks and user objectives. In other words we determine whether the software is suitable for helping the user execute his or her intended tasks.

It must be remembered that (software) products are never the goal in itself. They are produced to support their users in doing their real job. Suitability of a product is about how the functionality of the product supports the tasks the users are performing.

An accountant's job is to keep accounts. This can be done entirely using pen and paper. The accountant may however use a computer system to help. When the accountant has entered the data, the system may be able to store them, calculate sums from them, and produce reports presenting the data on the screen and on paper.

If the computer system does not calculate the needed sums, the suitability is lower than if it did.



Usually the largest part by far of the requirements is requirements belonging to the suitability attribute.

The basic way all software products work is to have and/or to allow entry of some data, to handle the data in appropriate ways, and to present the data and the handling results.



The very short story about what detailed suitability could concern is:

- ▶ Data availability (i.e., how do we get data, both in terms of background or reference data and/or data from other system information and in terms of data input and change facilities and the associated levels of data validations)
- ▶ Data handling (i.e., what is data used for, for example, as event-driven interrupts or signals and—not least: calculations, actions, and so forth based on data and other input)
- ▶ Result presentation (i.e., output facilities, for example, in terms of windows and reports)

The list is by no means exhaustive. It should, however, give an idea of how functionality requirements may be structured. Requirements tools, such as UML, provide help in expressing the suitability requirements.



A few examples of suitability requirements could be:

[D.72] The product shall contain a list of all postal codes in the United Kingdom.

[K.397] A postal code must be entered as part of an address.

[K.398] The postal code may be typed directly or selected from the list of postal codes.

Suitability is often expressed in use cases, because use cases provide an excellent way of describing what the users' tasks are and how the software product should support these.

Suitability testing can take place at all testing levels. All of the techniques discussed in Chapter 4 may be used in suitability testing, though use case testing seems to be the best.

ISO 9126 references ISO 9241-10 and states that its definition of suitability corresponds to suitability for the task in that standard. ISO 9241-10 defines that suitability for the task means that the dialogue should be suitable for the user's task and skill level.

This implies that suitability is closely related to operability, a subattribute of usability discussed in Section 5.1.2.



### 5.1.1.2 Accuracy Testing

In accuracy testing we test the requirements or needs concerned with the product's ability to provide the right or agreed upon results or effects.

A more detailed accuracy specification could concern:

- ▶ Algorithmic accuracy: Calculation of a value from other values and the correctness of function representation
- ▶ Calculation precision: Precision of calculated values
- ▶ Time accuracy: Accuracy of time related functionality
- ▶ Time precision: Precision of time related functionality



Accuracy requirements are often implied. If in doubt about these attributes during testing it is better to ask, rather than to assume, especially if you are not an absolute domain expert.



A few examples of accuracy requirements could be:

[230] During calculation all money values shall be rounded; except in the Japanese version, where money values always shall be truncated.

[232] All money calculations in euros shall be performed with three decimals.

[233] All money calculations in currencies other than euros shall be performed with two decimals.

[234] The system shall present all values for money with two decimals on the user interface and in reports.

Accuracy can be tested at all testing levels, the earlier the better. Some accuracy testing may even take place as static tests of design and code. Many of the techniques discussed in Chapter 4 may be used in accuracy testing.



### 5.1.1.3 Interoperability Testing

In interoperability testing we test the requirements or needs concerned with the ability of our software system to interact with other specified systems.

No software system stands alone; it will always have to interact with other systems in the intended deployment environment, such as hardware, other software systems like operating systems, database systems, browsers, and be-spoken systems, external data repositories, and network facilities.

The interoperability attributes are concerned with the specifications of all the interfaces the software system has to the external world at the time of deployment. The external systems may be part of the product being produced, or already existing products that we need to interface with.

Detailed interoperability could concern:

- ▶ Inbound interoperability: Ability to use output from standard, third party, or in-house products as input
- ▶ Outbound interoperability: Ability to produce output in the format used by standard, third-party, or in-house products
- ▶ Spawnability: Ability to activate other products
- ▶ Activatability: Ability to be activated by other products

It can be a huge task to test the interoperability because of the sheer number of possible combinations of interfaces for a system. It is often practically and/or economically impossible to achieve full coverage of all possible combinations. The Allpairs testing technique discussed in Section 4.1.7 can be used to select combinations. The intercomponent testing technique discussed in Section 4.2.9 is useful for designing test cases for interoperability testing.



A few examples of high-level interoperability requirements could be:

[I.509] The system shall obtain a record of a patient's hospitalization history from the central health register.

[I.87] When a patient is discharged, his or her hospitalization history shall be updated in the central health register.

[I.4] A discharge letter shall be produced by the letter-writer module when a patient is discharged.



Interoperability testing usually takes place at the system integration testing level.



Interoperability should not be confused with adaptability or replaceability, discussed in Section 5.2.6.

#### 5.1.1.4 Functional Security Testing

In security testing we test the requirements or needs concerned with the ability to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information, or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.

Detailed security could be concerning:

- ▶ Activity auditability: Log facilities for activities, actors, and so forth
- ▶ Accessability: Access control mechanisms
- ▶ Self-protectiveness: Ability to resist deliberate attempts to violate access control mechanisms
- ▶ Confinement: Ability to avoid accidental unauthorized access to facilities outside the application
- ▶ Protectiveness: Ability to resist deliberate attempts to access unauthorized facilities outside the application
- ▶ Data integrity: Protection against deliberate damage of data
- ▶ Data privacy: Protection of data against unauthorized access

Security testing can be split into functional security testing and technical security testing. In functional security testing we test the fulfillment of security attributes that can be explicitly expressed in requirements. These are typically requirements pertaining to the two first and the last of the attributes listed earlier.

**Ex.**

A few examples of functional security requirements could be:

[623] The system shall ensure that each registered user is member of at least one of the specified user groups.

[65] The system shall ensure that only users with delete privilege may discharge patients.

[72] The system shall ensure that only users with extended read privilege may see the full hospitalization history for a patient.

”

Functional security testing can be performed at all testing levels, again the earlier the better, also using static test of design and code. Many of the techniques discussed in Chapter 4 may be used in functional security testing, not the least of which are the defect-based techniques discussed in Section 4.3 and the experience-based techniques discussed in Section 4.4.

## 5.1.2 Usability Testing

Usability is the suitability of the software for its users, in terms of the effectiveness, efficiency, and satisfaction with which specified users can achieve specified goals in particular environments or contexts of use.

The *effectiveness* of a software product is its capability to enable users to achieve specified goals with accuracy and completeness. The *efficiency* of a product is its capability to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved. The *satisfaction* of a product is its capability to satisfy users.



### 5.1.2.1 Users Concerned with Usability

The usability attribute is related to users. It is important to get a complete overview of potential user groups and to take any kind of user characteristics into account when working with usability.

A user group for a product is a group of people who will be affected in similar ways by the product. A user group is not just the people entering data into the product and looking at the screen, though this is certainly an important user group. This group may indeed be divided into frequent users, occasional users, and rare users, or other relevant subgroups. User groups may also consist of those in charge of installing the product and those monitoring and maintaining it. Groups may be those getting information from the product, for example, in the form of reports or letters; and it may be those having to be near the product without actually interfering with it.

The list of appropriate user groups is of course very product-sensitive, and care should always be taken not to forget a potential user group.



For each of the user groups it is necessary to look at different characteristics. This could, for example, include:

- ▶ Age (e.g., preschool, children, teens, young adults, mature adults, and elderly)
- ▶ Attitude (e.g., hostile, neutral, enthusiastic)
- ▶ Cultural background
- ▶ Education (e.g., no education yet, illiterate, basic education, middle education, workman, university education)
- ▶ Disabilities (e.g., people who are dyslexic, color-blind, blind, partially sighted, deaf, mobility-impaired, or cognitively disabled)
- ▶ Gender
- ▶ Intelligence

### 5.1.2.2 Usability Subattributes

The ISO 9126 standard classifies usability as a nonfunctional quality attribute. It has got to do with how the functionality presents itself to the users. However, there is more to it than meets the eye; usability covers much more than just the look and feel of the product.

ISO 9126 breaks the usability attribute into the following subattributes:



- ▶ Understandability
- ▶ Learnability
- ▶ Operability
- ▶ Attractiveness

*Understandability* has got to do with how difficult it is to recognize the logical concept and find out how to apply it in practice. This may cover the:

- ▶ Extent to which the system maps the concepts employed in the business procedures
- ▶ Extent to which existing nomenclature is used
- ▶ Nature and presentation of structure of entities to work with
- ▶ Presentation of connections between entities

*Learnability* concerns the learning curve for the product. This may cover the:

- ▶ Extent to which a user of the system can learn how to use the system without external instruction
- ▶ Presence and nature of on-line help facilities for specified parts of the system
- ▶ Presence and nature of off-line help facilities for specified parts of the system
- ▶ Presence and nature of specific manuals

*Operability* is about what the product is like to use and control in deployment. This may cover the:

- ▶ Presence and nature of facilities for interactions with the product
- ▶ Consistency of the man-machine interface
- ▶ Presence, nature, and ordering of elements on each form
- ▶ Presence and nature of input and output formats
- ▶ Presence and nature of means of corrections of input
- ▶ Presence and nature of navigational means
- ▶ Number of operations and/or forms needed to perform a specified task

- ▶ Format, contents, and presentation of warnings and error messages
- ▶ Presence and nature of informative messages
- ▶ Pattern of human operational errors over stated periods of time under stated operational profiles according to defined reliability models

*Attractiveness* has got to do with how the users like the system and what may make them choose to acquire it in the first place. This may cover the:

- ▶ Use of colors
- ▶ Use of fonts
- ▶ Use of design elements, such as drawings and pictures
- ▶ Use of music and sounds
- ▶ Use of voices (male and female), languages, and accents
- ▶ Layout of user interfaces and reports
- ▶ Presence and nature of nontechnical documentation material
- ▶ Presence and nature of technical documentation material
- ▶ Presence and nature of specified demonstration facilities
- ▶ Presence and nature of marketing material

### 5.1.2.3 Accessibility

In recent years there has been more and more focus on equal opportunities, not the least of which are for people with disabilities. This also concerns software systems, which must be accessible and operable for everybody.

Rules are, for example, expressed in the Disability Discrimination Act covering United Kingdom and Australia, and Section 508 for the United States.

**Ex.**

A special and important attribute for usability of a product is therefore accessibility, even though this is not explicitly mentioned in ISO 9126.

In this context accessibility is the ease with which people with disabilities can operate the product.

Accessibility may cover the:

- ▶ Use of colors, especially mixtures of red and green
- ▶ Possibility of connecting special facilities, such as speaker reading the text aloud, Braille keyboard, voice recognition, and touch screens
- ▶ Possibility of using the product entirely by key strokes and/or voice commands
- ▶ Facilities for multiple key pressure using only one finger or other pointing device
- ▶ Possibility of enlarging forms and/or fonts
- ▶ Navigation consistency

A number of standards cover various aspects of accessibility aspects, including Web Contents Accessibility Guidelines from the World Wide Web Consortium (W3C), an international consortium working on Web standards.

#### 5.1.2.4 Establishing Usability Requirements

Like all other requirements, usability requirements should be expressed as explicitly as possible. Usability requirements can be derived from usability assessments (usually called by the very misleading name usability test, and also known as formative evaluation).



Usability assessment is a requirements elicitation technique—and it should be performed early, not on the finished product.

A usability assessment is performed by representative users who are given tasks to complete on a prototype of the products. This can be hand-drawn sketches of forms or mock-ups of the forms made in, for example, PowerPoint. Any thoughts and difficulties the users have in completing the tasks are recorded. This is best done if the users can be made to “think aloud” during the assessment.

After the usability assessment the comments are analyzed; the prototype may be changed and assessed again; and finally the usability requirements are derived.

Usability assessments can be done very primitively by review of prototypes and storyboards, or very sophisticatedly in purpose-built usability labs with two-way mirrors and video equipment.

People with different skills, for example, specialists in sociology, psychology, and ergonomics, may participate in the elicitation and documentation of usability requirements, specific standards, or special considerations to be made.

The usability requirements must be measurable. A requirement like this:



The user interface shall be nice to look at

is seen all too often, but it is no good.

**Ex.**

Here are a few examples of measurable usability requirements:

{UR.518} At least 95% of the primary users of the product shall answer either “very good” or “good,” when asked about their opinion of the look and feel of the user interface in the survey to be carried out two months after deployment.

{UR.523} At least 80% of estate agents with a minimum of five years experience shall be able to complete the task described in the use case {UC.78} in less than 30 minutes after 20 minutes instruction.

{UR.542} All push buttons shall be placed right aligned at the bottom of the forms.

{UR.557} All forms shall have an online help facility describing the purpose and the syntax of all the fields on the form.

{UR.516} All tasks described in Section 4.1 shall be completable with a maximum of five clicks.

Note that these usability requirements are nonfunctional, or functionality-sustaining. They cannot be expressed independently of functionality but must refer to the functionality to which they apply.

### 5.1.2.5 Testing Usability

Usability may be tested in various ways during the development life cycle. Techniques to use may be:

- ▶ Static tests
- ▶ Verification and validation of the implementation
- ▶ Surveys and questionnaires

*Static tests* can be performed as reviews and inspections of usability specifications. These may include so-called heuristic evaluation, where the design of the user interface is verified against recognized usability principles. Static testing finds defects early and is hence very cost-effective, not least of all for usability where mistakes in the user interface may be very expensive to remedy late in the development life cycle. Static testing is further discussed in Chapter 6.

The *verification and validation* of the implementation of the usability requirements are performed on the working system. Here the focus is on the usability requirements associated with, but not identical to, the functional requirements. Test procedures, use cases, or scenarios may be used to express what is to be done, whereas the actual usability testing is about whether the usability requirements are fulfilled.

The usability requirements may be in the form of requirements stated in natural languages, as the examples above show, but they may also be expressed in terms of prototypes or drawings. These may be more difficult to test against, but it is important to verify that an implementation is in fact reflecting the prototype agreed on by the future users.

In this form of usability testing it is particularly important that the test environment reflects the operational environment, not least of all in terms of space, light, noise, and other disturbing factors.

Coverage may be measured using the usability requirements as the coverage element.

The ultimate validation is the user acceptance test where the finished product should be accepted by the users as being the system that fulfills their requirements, expectations, and needs. Obviously great care should be taken all along the development to ensure that acceptance may actually be the results of the acceptance testing. Serious defects and failures identified at this point of time may turn out to be very expensive.

Other tests like syntax tests of input fields and tests of messages to users may be combined with usability, even though they (in a strictly ISO 9126-speaking sense) belong to functional testing.

Not all requirements can be tested in a static and dynamic testing. *Surveys and questionnaires* may be used where subjective measures, such as the percentage of representative future users who like or dislike the user interface, are needed.

The questions must be worded to reflect what we want to know about the users' feelings towards the product. We can make our own, or we may use standardized surveys such as SUMI or WAMMI.

SUMI, *The Software Usability Measurement Inventory*, is a tested and proven method of measuring software quality from the end user's point of view. It can assist with the detection of usability flaws before a product is shipped, and it is backed by an extensive reference database embedded in an effective analysis and report-generation tool. SUMI provides concrete measurements of usability, and these may be used as inspiration for usability requirements or completion criteria. See more on <http://sumi.ucc.ie>.

WAMMI is a Web analytics service to help Web site owners accomplish their business goals by measuring and tracking user reactions to Web site ease of use. See more on <http://www.wammi.com>.



## 5.2 Quality Attributes for Technical Test Analysts



As important as the functionality of a product may be, it cannot stand alone. The functionality will always behave and present itself in certain ways. This is what we call the nonfunctional or functionality-sustaining attributes of the product.



Historically these attributes have been neglected when requirements have been specified, and testers have "tested" some of the nonfunctional quality attributes based on their experience. This testing was very often just a negative test: The basic idea was to get the product to fail to see how much it could cope with without knowing what the needs and expectations were.

This ought not to happen. Nonfunctional requirements should be defined for all the functionality for the product in the requirements specification.

There are many suggestions for what nonfunctional requirements should cover.

Some classics standards, which have been around for quite some time, are listed here with the quality attributes they include:

<i>ISO 9126</i>	Functionality, reliability, usability, efficiency, maintainability, portability	
<i>McCall and Matsumoto</i>	Integrity, correctness, reliability, usability, efficiency, maintainability, testability, flexibility, portability, interoperability, reusability	
<i>IEEE 830</i>	Performance, reliability, availability, security, maintainability, portability	
<i>ESA PSS-05</i>	Performance, documentation, quality, safety, reliability, maintainability	

A working group under British Computer Society is currently working on a new standard for nonfunctional quality attributes. This includes the following:

<i>BCS working group</i>	memory management, performance/stress, procedure, reliability, security, interoperability, usability, portability, compatibility, maintainability, recovery, installability, configuration, disaster recovery, conversion
--------------------------	---

More information about this work can be found at:  
[www.testingstandards.co.uk](http://www.testingstandards.co.uk)



The nonfunctional requirement types covered in this section are:

- ▶ Reliability
- ▶ Efficiency
- ▶ Maintainability
- ▶ Portability



in accordance with ISO 9126. Usability testing is covered in the previous section, since it is of interest to test analysts.

Furthermore the technical aspects of security testing are covered here.

### 5.2.1 Technical Testing in General

”

In principle the nonfunctional testing is identical to the functional testing; it should be based on requirements and needs and use the test case design techniques discussed in Chapter 4.

However, testers can, and should, help developers and analysts define these requirements from the beginning; and we can review the requirements to ensure that they are comprehensive and testable.

It is important that the nonfunctional requirements are measurable and testable. This is, however, not always easy, at least not in the beginning. All too many nonfunctional requirements are expressed using words like “good,” “fast,” or “most of the time.”

To overcome this it must be remembered that each nonfunctional requirement must be expressed using a scale, a specific goal, and possibly also acceptable limits. The circumstances under which the goal is to be achieved must also be specified. Further information, for example, stretch, achieved records, and future goal, could also be given to put the specified goal in perspective.

**Ex.**

Let's look at an example. First a typical way of expressing a performance expectation:

“Reports must not take too long to be created.”

This should make you wonder which reports we are talking about, what “too long” is, and if this is a general expectation no matter the circumstances.

The requirements could be reworded to:

“[P.65] The creation of a full report of all the clients as specified in Req. [F.89] shall not take more than 15 minutes if launched between 8:00 and 16:00 on normal weekdays.”

This is much better. Now we know which report we are talking about, namely the one specified in the functional requirement [F.89]; we know that 15 minutes is acceptable, and we know that this is the expectation for a normal working day. The requirement could be even more specific, but the improvement in testability is already significant.



It can become a sport to dig behind imprecise nonfunctional expectations and find out what the real need is.

Apart from assisting during the requirements specification, technical testers must test the actual implementation of the nonfunctional requirements. This can be done at different testing levels depending on the types of requirements.

The coverage for the nonfunctional testing can be measured using non-functional requirements as the coverage element.

The testing of the nonfunctional quality attributes must be executed in a realistic environment reflecting the specified circumstances. This can be in terms of, for example, hardware, network, other systems, timing, place, load patterns, and operational profiles.

An operational profile is a description of:

- ▶ How many
- ▶ Of which user groups
- ▶ Will use what parts of the system
- ▶ When
- ▶ How much and/or how often



If care is not taken to ensure realistic circumstances the testing may be a complete waste of time and, even worse, create a false sense of confidence in the product.



### 5.2.1.1 Random Input Technique

In order to perform much of the technical testing to be discussed below, we have to set up test cases matching the operational profiles we are going to test and measure under. This means that in principle we have to define the same statistical distributions of input as those defined in the requirements.

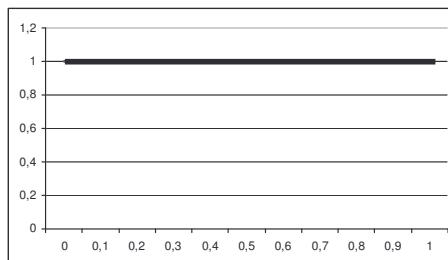
The random input technique can assist in generating input data based on a model of the input domain that defines all possible input values and their operational distribution.



Random input follows the input distribution; the input values are constrained and guided by this.

Expected input patterns can be estimated or may be known before deployment, and that knowledge can be used in testing. There are many possible distributions, but the most common ones are the uniform distribution and the normal distribution.

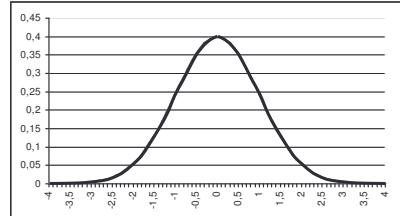
In uniform distribution the probability of each value in the value domain is equal.



**Ex.**

The outcome of throwing a die follows an equal distribution. The probability of getting a 6 is the same as that of getting any of the other values.

In the normal distribution there is an average value, which we have a high probability of getting. The further away from the average a value is, the lower is the possibility of getting that value.

**Ex.**

This is, for example, the case for the height of men in their forties. Most of these men are 1.80 meters. Only a few men stand 2 meters tall, and likewise only few reach no more than 1.6 meters.

For random testing we select input values for the test cases randomly from the input domain according to the input distribution.



If the distribution is unknown, we can always use a uniform distribution.

The pitfall of random input generation is that it can be very cumbersome to determine what the expected results of the test cases are. Random input is mainly interesting when the objective is to get the system to crash. A large number of test cases can be generated quite quickly, and long sequences of input can be run.

The expected result and the actual result are, however, usually not that important when we are performing reliability or performance testing.

The benefit of random input is that it is very cost-effective, especially if automated; the input to the test cases is cheap to develop (though the expected results may not be), the input requires little maintenance, and it gets around in the system in a trustworthy way.

Another benefit is that random input testing may find “unexpected” combinations and sequences and may detect initialization problems. It usually gives high code coverage. If automated it is a very persistent testing, and long test runs may also find resource problems, such as memory leaks or list overflows.

### 5.2.2 Technical Security Testing

In security testing we test the requirements or needs concerned with the ability to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information, or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.

Detailed security attributes may be:

- ▶ Activity auditability: Log facilities for activities, actors, and so on
- ▶ Accessability: Access control mechanisms
- ▶ Self-protectiveness: Ability to resist deliberate attempts to violate access control mechanisms
- ▶ Confinement: Ability to avoid accidental unauthorized access to facilities outside the application
- ▶ Protectiveness: Ability to resist deliberate attempts to access unauthorized facilities outside the application
- ▶ Data integrity: Protection against deliberate damage of data
- ▶ Data privacy: Protection of data against unauthorized access

Security testing can be split into functional security testing and technical security testing. In functional security testing we test the fulfillment of security attributes that can be explicitly expressed in requirements. In technical security testing we take on a much broader perspective.

It is impossible to express all technical security issues as testable requirements; there are simply too many ways in which things can go wrong and too many ways in which people with dishonest intentions can try to get to our valuables.

The valuables we have in software systems are data, both in the form of data and in the form of running code. This is what we need to protect, and technical security testing is about finding out if the system is able to withstand threats to the data.

Data can be jeopardized in a number of ways, typically:

- ▶ Read to obtain other valuables (e.g., credit card numbers or sensitive data)
- ▶ Copied for use without paying (e.g., music or entire products)
- ▶ Added for harmful effects (e.g., viruses or access requests)
- ▶ Changed (e.g., sensitive information or code instructions)
- ▶ Deleted (e.g., data)

The difficulty with technical security testing is that most of the testing techniques discussed in Chapter 4 usually have very little probability of finding security defects. The majority of the security defects are not defects in the sense that the product does not fulfill stated requirements or expectations in a narrow sense. The problems arise from the product having or allowing functionality that is not wanted by the future users and hence not specified, but implemented to ease the implementation without regard to the possible security side effect such an implementation might have.



**Ex.**

In a product protected by user identities and passwords, the passwords are normally stored in an encrypted way. As long as a user is logged on, however, the password is stored in an unencrypted way in order to obtain a better performance.

Already at the design of the system care should be taken to reduce the risks of these things happening. Designers need to take on the most pessimistic and malicious minds they possibly can. This is not as easy as it sounds. Most people trust their neighbor to a large extent and find it difficult to think of ways in which they themselves may be cheated or threatened. This is why some professional hackers who have changed their ways may be employed as security consultants.



Note that there may be conflicts between security requirements and performance requirements, in the sense that higher security may cause lower performance.

Testers can perform static tests on design and code implementation to look for vulnerabilities in the system at an early stage. Things to look out for may include:

- ▶ Use of temporary storage of sensitive data or information supposed to be kept secret (for example, passwords or encryption keys)
- ▶ Possible buffer overflows, both in input and internal data handling
- ▶ Exposure of sensitive data for example over networks
- ▶ Acceptance of unvalidated data either via a user interface or an external interface



The list is by no means exhaustive.

A special security issue is logical bombs or so-called Easter eggs, where harmful code has been written into the components during development. Static testing is the only technique that may find this.

**Ex.**

Examples of logical bombs may be:

An IF clause that is only true on specific data, where data might be erased.  
An IF clause that is only true for a specific bank account number, to which an extra amount of money is debited.



The testing technique of attacks, discussed in Section 4.4.4 is very useful for technical security testing. Checklists of effective attacks should be kept up-to-date.

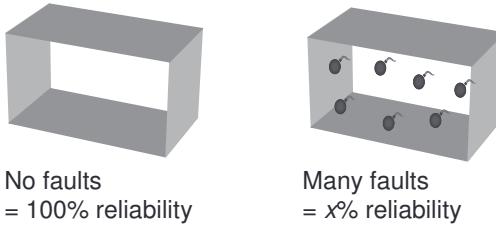
A more systematic approach perhaps is one supported by tools. A profile of the product in term of versions of operating software and middleware, identifications of developers and users, and details about internal networks is created by the use of a tool. Based on this, tools can scan the product for

known vulnerabilities, and this information can be used to develop targeted attack plans.

It is not possible to measure coverage for technical security testing, except in the form of items on checklists covered by a test.

### 5.2.3 Reliability Testing

Reliability is the probability that software will not cause the failure of a system for a specified time under specified conditions.



Concerning reliability we have got to be realistic: It is impossible to produce 100% fault-free products!

Functional testing and reliability testing are connected. The goal of functional testing is to obtain the highest possible reliability of the product within the given limits. The goal of reliability testing is to evaluate the reliability we have obtained.

The test object is the complete product. The reliability testing should be based on operational profiles specified for the product, and reliability goals expressed in requirements. This is not always easy or efficient to set up. In fact it is sometimes impossible to perform reliability testing before the product is in operation. Usually this is acceptable, and the reliability test will be part of the final acceptance test.

A specific reliability may be used as a test completion or test exit criterion for the system testing, allowing for earlier reliability testing.

During reliability testing it is essential to collect measurements for the evaluation. Failures are registered when they appear “spontaneously,” and they must be categorized and countable. We must also collect other measurements as appropriate, such as time and number of transactions.

The test must go on until “reliable” data has been obtained. This can take quite a while, certainly days and maybe even weeks or months.

The ISO 9126 standard breaks the reliability attributes down into a number of subattributes. There are:

- ▶ Maturity
- ▶ Fault tolerance (robustness)
- ▶ Recoverability



The standard explains each of them. These explanations are primarily intended as inspiration for nonfunctional requirements. They can also be used as checklists for static testing of reliability requirements, design, and implementation, and as inspiration for checklist-based reliability testing.

### 5.2.3.1 Maturity Testing

Maturity is the frequency of failures as a result of faults in the software.

A product's expected maturity is often expressed in terms of

- ▶ Mean time between failures (MTBF)
- ▶ Failures per test hour
- ▶ Failures per production time (typically months)
- ▶ Failures per number of transactions

The metrics may be further detailed by categorizing the accepted types of failures, for example, by severity.

**Ex.**

A few examples of reliability requirements are:

- [34] The MTBF for the product shall be more than one month on average in the first year of production.
- [72] The product shall have no more than two failures of severity 1 reported in the first six months of production.
- [281] The product shall have less than three failures per 10,000 transactions.

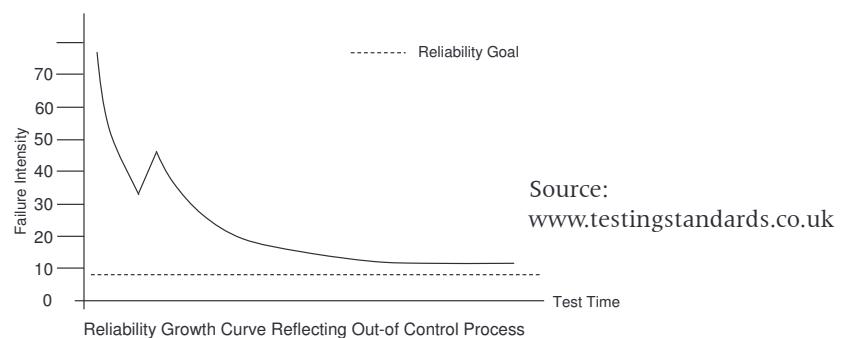
**Ex.**

For testing completion criteria we may have reliability goals like:

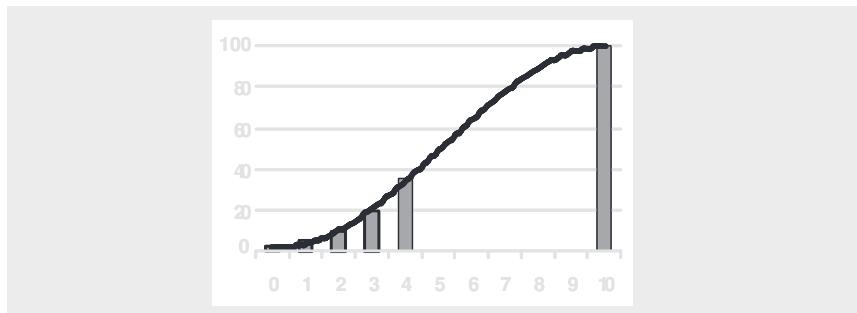
The testing can stop when less than one failure of severity 1 has been found during 20 hours of testing on average over at least two weeks of testing.

The results of the reliability testing can be graphic presentations of the measurements using a reliability growth model.

The following figure shows a reliability growth curve for a product where the reliability is getting better, but not achieving its goal.



Another way to present reliability data is in an S-curve. Here the total number of failures found over time is plotted against time.



The figure shows an S-curve for the total number of failures for each test week. The expected number of failures after 10 weeks' testing is 100. So far it looks as if the pattern of failures found follows the expectations.

S-curves are often used to determine when to stop the test. They are also useful for predicting when a reliability level will be reached. Furthermore, S-curves may be used to demonstrate the impact on reliability of a decision to deliver the software NOW!

In cases where late or complete reliability testing is not acceptable we must either reword the requirements, use statistical proof, or use analysis.



Reliability evaluation or reliability estimation is an activity where we analyze the fault-finding curves we have produced. We extrapolate from the curves and predict how many defects are left in the product.

Another way to predict remaining defects is to use estimation models based on the structure of the program, for example, knowledge of the size, the complexity, and the data transactions.

### 5.2.3.2 Robustness Testing

Fault tolerance or robustness is the product's ability to maintain a specified level of performance in the presence of software defects or infringement of a specified interface.

This may cover the product's:

- ▶ Containment of defects to specified parts of the system
- ▶ Reactions to failures of a given severity
- ▶ Self-monitoring of the operations and self-identification of defects
- ▶ Ability to allow specified work to continue after a failure of specified severity for specified parts of the system under specified conditions
- ▶ Loss of specified operations (functionality requirement or set of functionality requirements) in case of failure of specified severities in specified periods of time for specified parts of the system

- ▶ Loss of specified data in case of failure of specified severities in specified periods of time for specified parts of the system

The failures to consider in technical robustness testing are those forced on the product from external sources. Failures due to internal defects should be handled in the functional testing.

Ex.

Failures from external sources could, for example, be caused by lack of external storage capacity, external data storage not found, external services not available, or lack of memory.

Robustness testing can, like the other technical tests, start at the requirements level, with testers reviewing robustness requirements.

Even more important is review of design. Design can be made more or less defensive, that is more or less robust to external circumstances. A simple, though often overlooked way to make systems more robust is to check the return code of all system routine calls and take action in the code when system call is unsuccessful.

Lack of memory may be due to memory leaks. Possible memory leaks could be found using dynamic analysis, for example, during component testing.

Testing robustness in system testing and/or acceptance testing may require the use of simulators or other tools to expose the product to external problems otherwise difficult to produce.

### 5.2.3.3 Recoverability Testing

Recoverability is the product's ability to reestablish its required level of performance and recover the data directly affected after a failure.

This may cover aspects like:

- ▶ Downtime after a failure of specified severities in specified periods of time for specified parts of the system
- ▶ Uptime during specified periods of time for specified parts of the system over a specified period of time
- ▶ Downtime during specified periods of time for specified parts of the system over a specified period of time
- ▶ Time to reestablish consistent data in case of failure causing inconsistent data
- ▶ Built-in backup facilities
- ▶ Need for duplication (standby machine)
- ▶ Redundancy in the software system
- ▶ Reporting of effects of a crash
- ▶ "Advice" in connection with restart

Downtime after a failure, including time to reestablish data, is often measured as mean time to repair (MTTR). This is closely linked to analyzability, discussed in Section 5.2.5.



### 5.2.4 Efficiency Testing

In efficiency testing we test the requirements or needs concerned with the product's ability to provide appropriate performance, relative to the amount of resources used, under stated conditions.

The ISO 9126 standard breaks the efficiency attributes down into the sub-attributes:

- ▶ Time behavior (performance)
- ▶ Resource utilization



The standard also explains each of these. The explanations are primarily intended as inspiration for related nonfunctional requirements. They can also be used as checklists for static testing of efficiency requirements, design, and implementation, and as inspiration for checklist-based efficiency testing.

#### 5.2.4.1 Performance Testing

Time behavior or performance consists of the expectations towards the product's ability to provide appropriate response and processing time and throughput rates when performing its functions under stated conditions.

In short, performance is concerned with how fast specified parts of the functionality are. This may cover requirements concerning:

- ▶ The response time for specified online tasks under specified conditions
- ▶ The elapsed time for transfer of specified data under specified circumstances
- ▶ The internal processing time (for example, in CPU cycles) for specified tasks under specified conditions
- ▶ The elapsed processing time for specified batch tasks under specified conditions

Expectations towards performance must be expressed in performance requirements.

[P.65] The creation of a full report of all the clients as specified in Req. {F.89} shall not take more than 15 minutes if launched between 8:00 and 16:00 on normal weekdays.



[P.72] The creation of a full report of all the clients as specified in Req. {F.89} shall not use more than 35% of the CPU time.

Performance testing can be very expensive and time-consuming to perform. It can be especially costly to establish the correct environment to test in.

It is important to be absolutely sure that the environment and conditions are correctly specified in performance requirements and correctly established for the performance testing, not least in terms of hardware, operating system, middleware, and concurrent use patterns or operational profiles.

This is something that should be specified in connection with the requirements specification in order for the performance requirements and hence the performance test to be reliable.

Many tools support test of performance. The tools can report on response times and execution times, for example, for database lookups or data transfer, and they can identify bottlenecks in the functions, either internally in the product and/or in the network for Web-based products. Tools are discussed in Chapter 9.

In principle performance test verifies that performance requirements are fulfilled. The coverage item is performance requirements, and the coverage is hence measured as the percentage of all the performance requirements tested in a test. As for all testing, failures must be reported, and retest and regression test must be performed after corrections.

### ***Load Testing***

Load testing is a special subtype of performance testing concerned with the product's behavior under specified load conditions.

Load has two aspects, namely:

- ▶ Multiuser with ordinary realistic numbers of users
- ▶ Large, though still realistic number (volume) of concurrent users

The loads that the product is expected to be able to handle and the applicable response times must be expressed in load requirements.

We need to think about load early on during requirements specification. It is important to strike a balance in the load requirements so that they are within reason, but still take the future into account.

Load testing can be quite expensive, and it is important not to go overboard. Sometimes we may have to question the requirements: Are they realistic or "wearing both belt and braces?" In order to keep the load testing expenses under control we should use risk analysis to prioritize and plan the testing tasks.

Tools may be used to generate and/or simulate loads.

Load testing and performance testing are sometimes related in the sense that the load can be part of the condition specifications for performance requirements.

### **Stress Testing**

Stress is an expression of the product's capability for handling extreme situations.

When planning requirements for stress handling it is a good idea to remember Murphy's law:

"If anything can go wrong, it will."

or

"The unthinkable sometimes happens anyway."

In stress-handling requirements we are confronting the risk of the system not being able to handle extreme situations. We are dealing with risks concerning people, money, data, and the environment, risks that will materialize if the system can't cope in a stress situation.

[87] The system shall not crash but issue an error message and stop execution after acknowledgment, if too much data is loaded in the data load described in requirement [DL931].



Stress handling is closely related to other nonfunctional areas. The relationship between reliability and stress is that stress looks at reliability in extreme situations. With respect to usability, stress is about handling failures with grace, so that the user is not left in the dark about what is happening. Stress in connection with load is concerned with peak load over a short span of time.

When working with stress-related requirements we will have to think about what can go wrong. For data this could be in situations with too much data, too little data, or faulty data. For usage it could be too many simultaneous users, extended use (same operation "umpteen" times, system to run without restart for many hours/days/months), or maybe dropping the system or part of the system on the floor. External events such as power failure may also cause stress in the system.

Stress-handling is particularly important for Web applications, such as e-products and e-business; in telecommunication, safety- and security-critical systems; and real-time systems.

Stress on a system can to a large extent be handled by defensive programming. This means that stress testing can start early with review of design and code.

Stress testing should have high "product" coverage. Even if some stress prevention works in one place it may not work in another part of the system.

Stress testing should be imaginative, but on the other hand it should not go overboard. When planning the stress test we need to make a risk analysis. Even the unthinkable may happen too rarely to warrant a test.



### ***Scalability Testing***

Scalability is the product's ability to meet future efficiency requirements. This is not really a test, since naturally we cannot perform it against existing requirements.

This technique is more a scalability assessment; what we are aiming at is finding out how the product reacts to growth in, for example, number of users or amounts in data.

The product's ability to keep its performance requirements may also be monitored on an ongoing basis during deployment to provide us with the opportunity to take action before the product may break under the load.

#### **5.2.4.2 Resource Utilization Testing**

Resource utilization is the expectations towards the product's use of appropriate amounts of resources when the software performs its functions, under stated conditions.

In short, resource utilization has got to do with what is used and what is needed. This may cover requirements concerning:

- ▶ The amount of CPU resources used for specified functions
- ▶ The amount of internal memory resources used for specified functions
- ▶ The amount of external memory resources used for specified functions
- ▶ Levels of memory leakage
- ▶ The presence, appropriateness, and availability of human resources, peripherals, external software, various material

In modern systems memory is rarely a problem, but it may be in, for example, games and also in real-time embedded systems. In the latter, memory usage, also referred to as memory footprint, may be the object of precise specification and thorough testing.



#### **5.2.5 Maintainability Testing**

In maintainability testing we test the requirements or needs concerned with the product's ability to be analyzed and modified. Modifications may include corrections of defects, improvements or adaptations of the software to changes in the environment, and enhancements in requirements and functional specifications.

The ISO 9126 standard breaks the efficiency attributes down into the sub-attributes:



- ▶ Analyzability
- ▶ Changeability

- ▶ Stability
- ▶ Testability

The standard also explains each of these. The explanations are primarily again intended as inspiration for related nonfunctional requirements and they may be used for checklists.

Maintainability testing can be performed as static testing, where the structure, complexity, and other attributes of the code and the documentation are reviewed or undergoing inspections based on the pertaining maintenance requirements. Static analysis of the code may also be used to ensure its adherence to coding standards and to obtain measurements, such as complexity measures.

The maintainability test may also be performed dynamically in the sense that specified maintenance procedures are executed and compared to pertaining requirements. What is measured in this type of maintenance testing is typically the effort involved in the maintenance activities.

The dynamic maintenance testing may be combined with other tests, typically functional testing, where the failures found and the underlying defects to be corrected may serve as those the maintainability procedures are tested with.

### 5.2.5.1 Analyzability Testing

Analyzability is the ability of maintainers to identify deficiencies, diagnose the cause of failures, and identify areas requiring modification to implement required changes. A way to measure analyzability is MTTR, mean time to repair.

Analyzability may cover aspects like:

- ▶ Understandability: Making the design documentation, including the source code, understood by maintainers
- ▶ Design standard compliance: Adherence to defined design standards
- ▶ Coding standard compliance: Adherence to defined coding standards
- ▶ Diagnosability: Presence and nature of diagnostic functions in the code
- ▶ Traceability: Presence of traces between elements, for example, between requirements and test cases, and requirements and design and code
- ▶ Technical manual helpfulness: Nature of any technical manual or specification



### 5.2.5.2 Changeability Testing

Changeability is the capability for implementation of a specified modification in the product.

Changeability may cover aspects like:

- ▶ Modularity: The structure of the software
- ▶ Code change efficiency: Capability for implementing required changes
- ▶ Documentation change efficiency: Capability for documenting implemented changes

### 5.2.5.3 Stability Testing

Stability is the capability of the product to avoid unexpected effects from modifications of the software, that is, to the risk of unexpected effects from modifications.

Stability may cover aspects like:

- ▶ Data cohesion: Usage of data structures
- ▶ Refailure rate: Pattern of new failures introduced as an effect of implementation of required changes

In other words, stability has got to do with the structure of the software and, not least, of the data. We might get an impression of the stability of the product during regression testing after defect correction.

Experience shows that in general 50% of the original number of defects remain in a product after defects are corrected. These are distributed like this:

- |  |     |
|--|-----|
| ▶ Original defects remaining                           | 20% |
| ▶ Existing defects revealed after correction of others | 10% |
| ▶ New defects introduced during defect correction      | 20% |

Stability has got to do with new defects introduced during defect correction, that is, how likely is that developers or maintenance staff accidentally make new mistakes and hence place new defects in the product when they are in fact engaged in correcting identified defects.

It is of course important for an organization to get its own measurements for the rate of new defect introduction and also for the effectiveness of defect correction.

### 5.2.5.4 Testability Testing

Testability is the capability of validating the modified system, that is, how easy it is to perform testing of changes, either new tests or confirmation test, and how easy it is to perform regression testing.

This is influenced both by the structure of the product itself and by the



structure of the test specification and other testware.

Testability is also influenced by how configuration management is performed. The better the control over the testware, not least the test data, and the product, and the documentation of the relationships between product versions and testware versions is, the better is the testability.



### 5.2.6 Portability Testing

In portability testing we test the requirements or needs concerned with the product's ability to be transferred into its intended environment. The environment may include the organization in which the product is used and the hardware, software, and network environment.

The porting may be the first porting from a development or test environment into a deployment environment, or it may be the porting from one deployment environment to another at a later point in time.

Portability is primarily an issue for software products or software subsystems, not so much for, for example, hardware subsystems. These are usually part of the environment into which the software subsystem or product is being ported.

The ISO 9126 standard breaks the portability attributes down into the sub-attributes:

- ▶ Installability
- ▶ Coexistence
- ▶ Adaptability
- ▶ Replaceability



The standard also explains each of these. The explanations are again primarily intended as inspiration for related nonfunctional requirements and for checklists.

#### 5.2.6.1 Installability Testing

Installability is the capability of installing the product in a specified environment.

Installability may cover aspects like:

- ▶ *Space demand*: Temporary space to be used during installation of the software in a specified environment.
- ▶ *Checking prerequisites*: Facilities to ensure that the target environment is meeting the demands of the product, for example, in terms of operating system, hardware, and middleware.
- ▶ *Installation procedures*: Existence and understandability of installation aids such as general or specific installation scripts, installation manuals, or wizards. This may also include requirements concerning the time and effort to be spent on the installation task.
- ▶ *Completeness*: Facilities for checking that an installation is complete,

- for example, in terms of checklists from configuration management
- ▶ *Installation interruption*: Possibility of interrupting an installation and rolling any work done back to leave the environment unchanged
  - ▶ *Customization*: The capability of setting or changing parameters at installation time in a specified environment.
  - ▶ *Initialization*: The capability of setting up initial information at installation time, both internal and external in a specified environment.
  - ▶ *Deinstallation*: Facilities for removing the product partly (downgrading) or completely from the environment.

#### 5.2.6.2 Coexistence Testing

Coexistence is the software product's capability to coexist with other independent software products in a common environment sharing common resources. Today with powerful servers, PCs, and portables and with more and more functionality in everything being controlled by software, the coexistence of systems is a growing issue.

**Ex.**

As cars began to have more and more software installed to control the functions of the car, a common joke was that the windshield wipers would only work if the passenger in the right backseat weighed less than 80 kilos.

This example may seem far out, but failed coexistence may have the strangest effects. Such failures may be caused by systems using the same area of memory and thus getting corrupted data, or by systems affecting each other's performance.

Coexistence can be very difficult to specify, since we don't always know into which environment our software product is being placed. Precautions can be taken in the form of resource utilization requirements, which may then be compared to resource utilization of any other systems our system is going to coexist with, and the available resources in the target environment.

Coexistence testing can also be very difficult to perform, since it is usually impossible to establish correct test environments for this. Often coexistence is tested after acceptance testing of the product and the installation in the target environment. This is obviously risky, and the decision to do so should be made based on a risk analysis.

#### 5.2.6.3 Adaptability

Adaptability is the capability of the software product to be adapted to different specified environments without applying actions or means other than those provided for this purpose for the system.

Products and systems are rarely permanent and unchangeable these days and it will often happen that a system our system interfaces with will have to

be replaced by a newer version or a completely different system. In this case our system will have to be adapted to interface with the new system in the environment instead of with the old one.

Ease of changing interfacing systems may be achieved by using communication standards, such as HTML, or by constructing the software in such a way that it can itself detect and adjust to external communication needs.

Adaptability is primarily of interest to organizations developing commercial off-the-shelf (COTS) products or systems of systems.

Adaptability may cover aspects like:

- ▶ *Hardware dependency*: Dependence on specific hardware for the system's adaptation to a different specified environment
- ▶ *Software dependency*: Dependence on specific external software for the system's adaptation to a different specified environment
- ▶ *Representation dependency*: Dependence on specific data representation for the system's adaptation to a different specified environment
- ▶ *Standard language conformance*: Conformance to the formal standard version of a programming language
- ▶ *Dependency encapsulation*: The isolation of dependent code from independent code
- ▶ *Text convertability*: The capability for converting text to fit a specified environment

#### 5.2.6.4 Replaceability Testing

Replaceability is the capability of the product to be used in place of another specified product for the same purpose in the same environment.

This is the opposite of adaptability, because in this case our system replaces an old one. The issues for adaptability therefore also apply for replaceability. There is also a certain overlap with installability.

Furthermore replaceability may cover aspects like:

- ▶ *Data loadability*: Facilities for loading existing data into permanent storage in our system
- ▶ *Data convertability*: Facilities for converting existing data to fit into our system

## Questions

1. What are the quality attributes defined by ISO 9126?
2. What are the subattributes under functionality?
3. What are the main concerns for suitability requirements and testing?
4. What do we need to be aware of concerning accuracy testing?

5. What does interoperability mean?
6. Why do we need to think about security?
7. What is a user group?
8. What are the subattributes for usability?
9. For whom is accessibility especially important?
10. How may some usability requirements be found?
11. What is the coverage element for usability testing?
12. What is SUMI?
13. Why is nonfunctional testing traditionally a focus point for testing?
14. What are sources for nonfunctional attributes other than ISO 9126?
15. How must nonfunctional requirements be expressed?
16. What is an operational profile?
17. What is random input creation?
18. How can random input be used in nonfunctional testing?
19. How may data be jeopardized?
20. Which other nonfunctional attribute may security affect?
21. What is reliability?
22. How is reliability testing different from other nonfunctional testing?
23. For how long must reliability testing go on?
24. How can reliability estimation be performed?
25. What are the reliability subattributes?
26. What will be defined for testing of these subattributes to be possible?
27. What are the subattributes for efficiency?
28. What is important for efficiency testing to be reliable?
29. What are load requirements about?
30. What is stress for a product?
31. What is scalability about?
32. What resources may a product use?
33. What are the subattributes for maintainability?
34. What is MTTR?
35. Why are not all defects removed when a product is returned for confirmation testing and regression testing?
36. What is portability?
37. What are the subattributes for portability?
38. What may the problems with coexistence of systems be?
39. To which other subattributes does replaceability relate?
40. How can adaptability be enhanced/supported?

## **Reviews (Static Testing)**

**S**tatic testing—or reviewing – in the sense of having a second pair of eyes looking at something one has produced has probably been practiced since the first cavemen painted the walls of the caves.

In software the techniques were formally introduced in the 1970s. Since then a large volume of articles and a few books and comprehensive studies about static testing have been published.

Despite this there is still some confusion about terminology. Are we talking about reviews or static testing?

In this book the term “static testing” is chosen as the overall term for the testing that is not dynamic (i.e., the testing where the product is not executed (and hence static)). However, beware; the syllabus uses the overall term of “review,” even though one of the types is in itself called review.

### **Contents**

- 6.1 General Principles for Static Testing
- 6.2 Static Testing Types
- 6.3 Static Testing in the Life Cycle
- 6.4 Introducing Static Testing

## **6.1 General Principles for Static Testing**

### **6.1.1 History of Static Testing**

In the software industry static testing was first officially introduced by Mr. Fagan in the early 1970s. He wrote: “We had a number of projects coming in late in a number of departments. I knew I had to take some action to reverse this trend, so I worked with several of my colleagues to perform some analyses of what was happening, and we found that there were defects in some of our designs that were causing the delays. In some cases we were 12 weeks behind schedule and way over budget. So we designed and implemented an organized process to search for design defects at a very early point in the development process and eliminated them before they could become a problem and cause delays and budget overruns.” Static testing as we know it today was born.



Static testing appears in the maturity models CMMI® and ISO15504 (SPICE) and also in ISO9000. In CMMI® static testing is represented at level 2 and 3 in the validation and verification process areas. The maturity models CMMI® and ISO 15504 are discussed in more detail in Chapter 8.

### 6.1.2 Static Testing Definition

Dynamic testing is testing of software where the object under testing, the code, is being executed on a computer. Dynamic testing requires something that is executable and a more or less elaborate test environment. Dynamic testing finds failures, that is, situations where the object under testing does not behave as expected.

In contrast to this, static testing is testing—or quality assurance—where the object under testing is not being executed on a computer. Static testing can be performed on anything that can be read throughout the product life cycle, and it requires no special investments.



The ISEB/ISTQB vocabulary defines static testing as: “Testing of a component or system at specification or implementation level without execution of that software (e.g., reviews or static code analysis).”



The IEEE 1028 Standard for Software Review and Audits provides a thorough definition and description of review and audit techniques. It defines review as: “An evaluation of software element(s) or project status to ascertain discrepancies from planned results or to recommend improvement. This evaluation follows a formal process (for example, management review, technical review, software inspection or walk-through).”



The primary objective of any static testing is to find defects in the object under testing and hence provide information about the quality of what is being produced. When we perform static testing we look directly at the written work products, the documents. That means that we are looking for defects—those that the producer happened to place there following a human mistake.

As can be seen from the IEEE 1028 definition there are a number of different static testing types or techniques. The types to be discussed here are, in order of formality:



- ▶ Informal reviews
- ▶ Walk-through
- ▶ Technical review
- ▶ Management review
- ▶ Inspection
- ▶ Audit

A quick overview of the similarities and differences in the most commonly used types is shown in the following table. The types are discussed in detail in the following sections.

	Walk-through	Technical review	Management review	Inspection
<b>Primary purpose</b>	Finding defects	Finding defects	Finding defects	Finding defects
<b>Secondary purpose</b>	Sharing knowledge	Make decisions	Monitor and control progress	Process improvement
<b>Preparation</b>	Usually none	Familiarization	Familiarization	Formal preparation
<b>Usage of basis</b>	Rarely	Maybe	Maybe	Always
<b>Leadership of meeting</b>	Author	As appropriate	As appropriate	Trained moderator
<b>Recommended group size</b>	2–7	3 or more	3 or more	3–6
<b>Formal procedure</b>	Usually not	Sometimes	Sometimes	Always
<b>Volume of material</b>	Relatively low	Moderate to high	Moderate to high	Relatively low
<b>Collection of metrics</b>	Usually not	Sometimes	Sometimes	Always
<b>Output</b>	Sometimes an informal report	More or less formal report	More or less formal report	Defect list, measurements, and formal report

### 6.1.2.1 Static Testing Objects

Everything that can be read can be the object of static testing. If we look at the documents being produced during the development and maintenance of a product, there are documents for static testing in every phase from the earliest conception of the product to its disposal. Also documents produced by the organization and the supporting process areas can, and should be, objects for static testing.



A few examples of documents that can be subjected to static testing are listed below. The list is by no means exhaustive and should only be used as inspiration for identifying possible static testing objects in your organization.

- ▶ Organizational documents, such as policies, strategies, plans, reports, sales and marketing material
- ▶ Plans for products, projects, quality assurance, test, configuration management, customer interface, supplier agreements
- ▶ Requirement specifications from the business and from the users, and for software, hardware, data, network, services, and so forth

- ▶ Design for product, architecture, detailed components
- ▶ The actual product, such as code, data, hardware, installation guides, manuals, educational material
- ▶ Test specifications and test reports for component testing, integration testing, system testing, acceptance test
- ▶ Process descriptions, templates, examples, technique descriptions

### 6.1.3 Static Testing Cost/Benefit



There are costs associated with static testing, mainly because it takes time to perform.

A rule of thumb says that 15% of the development budget should be reserved for all the static testing activities.

These include:

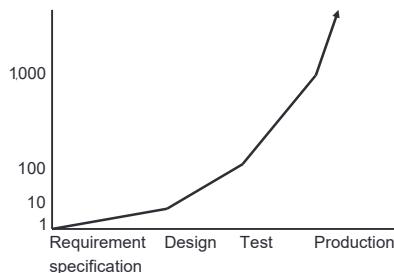
- ▶ Description of the static testing processes in general
- ▶ Performance of static testing
- ▶ Collection and analysis of metrics
- ▶ Improvement of the static testing processes



*On the other hand the benefits of static testing are plenty.*

First of all, the benefit lies in the early defect detection. We can save a lot of money by finding the defects close to their introduction.

Original from Grove Consultants – also inspired by IBM.



The graph above shows that the cost of correction of a defect grows with a factor 10 for each development phase it “survives.”

If a defect is introduced in the requirements and found and corrected in the requirements phase, we can say that that costs us 1 unit. If the defect is overlooked and not found until during design, it will cost us 10 units to correct. The cost grows as illustrated, and if the defect is not found before it causes a failure when the product is in use, it might cost us 1,000 units or even more.

If we add the fact that research shows that more than half of the defects found in the lifetime of a product—from requirements to disposal—can be traced back to defects in the requirements, it is clear that there is money to

be saved by performing static testing from the very start of a project. Part of the cost involved in defect correction is the time it takes to actually locate the defect. Since we find the defects directly in static testing, not through failures as in dynamic testing, it is obviously faster to identify what to change.

Early defect detection also gives other benefits to the organization:

- ▶ Management gets an earlier insight into the quality of the product, if we report information about our early findings.
- ▶ We get better productivity in development and an overall increase in efficiency, partly because defects are found early, partly because static testing gives better insight into what is to be produced.
- ▶ Our dynamic testing time is decreased because we have fewer defects left to deal with.
- ▶ Performance of static testing also means that fewer defects are sent out to the customer, and this gives a higher trustworthiness and less maintenance.
- ▶ On the softer side static testing can spring off new ideas on how to do things better, because knowledge sharing is an important part of static testing.
- ▶ A better esprit de corps can be promoted, since more stakeholders are involved in getting the best quality of the product from the beginning—we did this together!



In more mature organizations, data collected from inspections or reviews can be used to improve the static testing process and any other process further. The costs of improvements to processes other than the static testing processes themselves are not included in the 15% cost of static testing.



#### 6.1.4 Static Testing Generic Process

Even though the static testing techniques are different in some ways, they should all be performed within the framework of the generic test process.

##### 6.1.4.1 Test Process Applied to Static Testing

All the static testing types have in common that their performance should follow the general test process as depicted here.

The process activities are:

- ▶ Planning and control
- ▶ Analysis and design
- ▶ Implementation and execution
- ▶ Evaluating exit criteria and reporting
- ▶ Test closure activities



Traditionally rework has been considered part of the static testing process.

This is in principle no longer the case, since defect correction in the modern understanding of testing belongs to the development or maintenance process, not the testing process.

The general test process applied to static testing includes the following activities, adjusted to the static testing:

- ▶ Planning and control
  - ▶ Quality criteria are defined
  - ▶ The participants are selected
  - ▶ The test meeting, if any, is planned
- ▶ Analysis and design
  - ▶ The material is distributed
  - ▶ The participants are briefed about the assignment
- ▶ Implementation and execution
  - ▶ Static testing execution, usually at the reviewers' own desks
  - ▶ The static testing results are collected
  - ▶ Metrics may be collected about the performance and the results
- ▶ Evaluating exit criteria and reporting
  - ▶ The results are evaluated against any exit criteria
  - ▶ Static testing report may be produced
- ▶ Test closure activities

#### 6.1.4.2 Static Testing Checking

The checking done in static testing can be done in three directions relative to the document under testing. The directions are:

- ▶ Backwards  
Here we check if the document corresponds to its basis documentation and other related documents.
- ▶ Internal  
Here we check if the document is kept within the bounds of its purpose and scope, conforms to the template or standard it is built upon, is understandable, and free of spelling mistakes, incomprehensible language, and unexplained abbreviations.
- ▶ Forwards  
Here we check if the document is useful as the basis for further (development) work.

Like dynamic testing, static testing requires the existence of basis documentation specifying what we are testing against and what the requirements for the document under testing are. If basis documentation isn't available we can only check the document internally, that is, check the document against itself, for example, for inconsistencies.

Without basis documentation our testing will be based on assumptions and feelings, and that is not testing.

Basis documentation can be other documents in the life cycle, checklists, or standards. Standards can be project-specific, company-specific, and/or public.



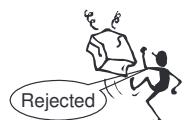
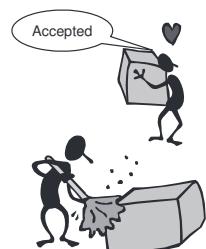
#### 6.1.4.3 Outcome of Static Testing

A static test on a particular document will have one of three possible outcomes, namely accepted, accepted with rework, or rejected.

(1) If everything is fine and no defects are found, the document can be accepted as it is. No further work is necessary before the document can be used as the basis for further work or be documented for other quality assurance activities.

(2) It may, on the other hand, happen that a number of defects are found. If the defects are not too many and not too serious compared with the quality criteria it can then be decided that some rework is needed before the document can be used further, but a restatic testing is not required. The document is accepted on the understanding that the necessary rework will be done before the document is released.

(3) In the case where the defects found are too numerous and/or too serious in view of the quality criteria, the document can be rejected. This means that it is returned to development for further work before it may be presented for static testing again.



#### 6.1.5 Roles in Static Testing

For static testing, as in dynamic testing, a number of roles to be filled are defined. The generic roles are:

- ▶ Decision maker
- ▶ Leader
- ▶ Author
- ▶ Appropriate staff to execute the static test
- ▶ Reader/presenter
- ▶ Recorder



Not all roles need necessarily be filled for all static testing types. More people may have identical roles in a static test, or one person may have several roles.

The decision maker is the person for whom a given static test is performed. The decision maker is the one to determine if the objectives of the static testing have been met. This role is only found in the more formal static test types.

The leader is the person responsible for the course of the static test, including administration, logistics, chairing of formal meetings, ensuring that objectives are met, and issuing of appropriate output. This role is only found in the more formal static testing types.

The author is the producer of the document to undergo the static test. Many of the roles in the development process can hence appear in the role of author depending on the document. The author may, for example, be:

- ▶ Project manager—for plans
- ▶ Test manager—for test plans
- ▶ Analyst—for requirements
- ▶ Designer—for designs
- ▶ Programmer—for code
- ▶ Test analyst—for test specifications
- ▶ Method engineer—for process descriptions

The appropriate staff to execute the static testing is comprised of reviewers or inspectors, depending on the type of static testing they are involved in. These are the people “executing” the static test, that is, those looking at the document. Technical as well as management staff may participate depending on the type of static testing in question.

The reader or presenter reads, presents, or paraphrases the document under static testing during the process.

The recorder records the issues raised during a static testing meeting. The person holding this role should not have other roles in the particular static test.

### 6.1.6 Static Testing Type(s) Selection

The static testing types have their individual strengths and weaknesses. In each particular case we have to choose between them and select the most suitable one or ones to use.

The selection can be based on a number of aspects, including:



- ▶ Risk analysis  
The higher the product risk is, the more formal the static testing type should be. Risks are discussed in Section 3.5
- ▶ Secondary objective  
The primary objective of static testing is always to find defects, but as shown in the table above and discussed below, each type has its own secondary objective. This could be used as a selection criterion if a specific secondary objective is wanted.

- ▶ Development phase

The earlier we are in the development, the more formal the static test should be to make sure that we are on the right track from the beginning.

The criticality of a document and the current development phase may be used as combined selection criteria. In general it can be said that the higher the criticality and the earlier in the development process, the more formal should the static testing types we use be, whereas the formality can be loosened for the lower criticality documents as we get further in to the development process.

Static testing types may be mixed. Using one static testing type is far better than none, and using several static testing types makes the static testing even stronger. Note that inspection should never be the first static testing type to apply, as this will jeopardize the effectiveness of the inspection.

Here are some examples of how static testing types may be mixed:

*Informal review -> Inspection*

This order ensures that the trivial defects have been removed before the inspection so that the inspection can be focused on major issues.

*Technical Review -> Inspection -> Walk-through*

This order ensures that the document is as defect free as we may expect and that it is ready for transfer to another group of people in the development. This other group gets the best starting point by being introduced to the document by the author.

*Technical review -> Walk-through*

This sequence of static testing types is less formal than the one above, but the objectives are the same.

*Walk-through -> Inspection -> Informal review*

This order ensures that the author is on the right track and can carry on working on the document until it is ready for inspection. After the inspection any minor spelling, grammar, and formatting issues will be caught before the document is released.

*Informal review -> Technical review -> Inspection*

This sequence is the most formal, and it ensures that the document doesn't have minor defects before the technical review and that the document is as defect free as we may expect both from a technical and a more formalistic point of view.

### 6.1.6.1 Static Testing of Code

Code is different from other types of documents in the sense that source code can be the object of static testing, as well as of static analysis and dynamic component testing.



All of these testing types are important because each type reveals different types of defects in the code.

## 6.2 Static Testing Types

The static testing types we are going to discuss in detail are, as mentioned earlier:

- ▶ Informal review
- ▶ Walk-through
- ▶ Technical review
- ▶ Management review
- ▶ Inspection
- ▶ Audit

### 6.2.1 Informal Review

The informal review is, as the name indicates, the least formal type of static test. This is what we all do all the time, mostly without thinking about it as a review: asking a colleague or a friend to look at something we have produced.

The objectives of informal reviews are very individual, depending on the author's needs. It could be everything from finding spelling and grammar mistakes, to the structure of the product, to the actual contents from a professional point of view.



An informal review follows no formal documented process. The participants are normally just the author and one or two reviewers. The reviewer(s) are usually chosen by the author. Most people have a network of reviewers to choose from, depending on the document to be reviewed and the review objective.



I have a number of reviewers for special objectives: one for spelling and grammar, one for formalities and references, and a couple of trusted colleagues for contents.

This type of static testing can be performed on any document at any state during its production. The author decides when he or she would like an informal review to take place. It could be on an early draft to make sure the structure and level are correct, or it could be on the final draft to iron out the last tiny wrinkles before the document is released.

The reviewer reviews the document when it fits into his or her schedule after agreement with the author. Basis documents are rarely used; the review is done according to the reviewer's perception.

The author typically gets feedback in either pure verbal form or in the form of notes scribbled on the document under testing.

Even though informal reviews rarely involve meetings as such, it is always a good idea to go through the notes with the reviewer(s). Notes may be unreadable and they may be difficult to understand for the author.



The result of an informal review varies and is very much dependent on the review skills of the chosen reviewer(s).

The author decides when the review is over. He or she may correct the document as he or she seems fit; there are no obligations following an informal review.

There are a few disadvantages connected with informal reviews. One is the dependency of the reviewers' reviewing skill, but that can easily be overcome by finding the right reviewers for the job. Another disadvantage is that there are usually no records kept of the reviews and hence no data available for calculation of effectiveness.

The benefits of informal reviews should, however, not be underestimated. Even though the reviews are informal they are very useful. In most companies no material must be delivered to another without having at least been through an informal review.



### 6.2.2 Walk-Through

A walk-through is a step-by-step presentation of a document by the author at a walk-through meeting. The primary objective is to find defects. Quite often the author discovers defects him- or herself just by going through the document.

The secondary objective is to create a common understanding of the contents of the document under testing. This is in fact often regarded as the primary objective, and it goes both ways. It is not necessarily just a question of the participants understanding what the author has thought; it can be a question of the author getting an understanding of where the participants want to go.



Walk-throughs are usually planned to take place at specific stages of the development. It can be early in the production process for a document to make sure the author is going in the right direction, or it may be as part of a handover of the objects to those who are going to use the document as the basis for their work.

Any document may be the object of a walk-through. The object is, however, most often code or design since a common understanding of the document is most important here.

The process for walk-throughs is usually not very formal. When the document to test is in a state corresponding to the defined entry criteria, the walk-through is scheduled and the participants, usually 3–7 people, are invited. The reviewers may get the document in advance to familiarize themselves with it, but there is no formal preparation required.

A walk-through meeting is always part of the process. The author acts as the presenter of the document and the only other role represented is reviewers (listeners). In cases of potential conflict a neutral facilitator may be present.



At the meeting the author “walks through” the object. This may be in the form of a dry run of the design or the code using scenarios or cases, or a step-by-step presentation of the contents. In the course of this the defects, omissions, possible changes, improvement ideas, style issues, and alternatives that pop up are noted and discussed.

Walk-through meetings should not last for more than 1–2 hours, so the volume that can be “walked through” cannot be too high. If the full volume of the document is too high, representative samples must be selected, and the information gathered must be applied to the rest of the document during rework.

After the walk-through meeting an informal report should be produced summarizing the findings, at least according to IEEE 1028.

The exit criteria for a walk-through are usually that the meeting has been held and the report approved. Corrections to the document under test are made at the author’s discretion.

The only slight disadvantage of walk-throughs is that the benefit depends on the author’s ability to present the object. Some people find it extremely hard to express themselves verbally in front of an audience and that may jeopardize the benefit of a walk-through. It is said that practice makes perfect, and this also applies to walk-throughs. If the author finds it hard, then start with a very small audience—maybe only one person and practice the technique.

Walk-throughs are well worth the effort. The defect finding is an important benefit, but the transfer and sharing of knowledge and understanding is even more important and useful in an organization. This is valid for groups of experienced people, groups of people with varied experience, and for groups with newcomers under training.



### 6.2.3 Technical Review

A technical review is a peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. A technical review is also known as a peer review.

Also here the primary objective is to find defects. The secondary objective is to make technical decisions and (one hopes) reach consensus about the approach to the work.

Technical reviews are usually planned to take place at certain times in the development life cycle.

Any technical, that is, nonmanagement, document can be the object of a technical review. Basis material in the form of preceding documents, requirements for the object, and/or standards and checklists should be used.

Technical reviews must have a manager, who is not the author. The manager decides if the document is ready for review based on the demands for a review expressed in the relevant policy and the test or quality assurance

plan.

The defined roles for technical reviews are fairly formal. The roles to be represented are a manager, the chairperson for the review meeting, the presenter, the reviewers, the author, and the recorder. The manager, chair, and presenter role is often filled by the same person. The total number of participants should be 3–10 people.

It is important that the participants in a technical review are more or less at the same level in the organization. A manager should not participate in a technical review, as this might make the author and perhaps other participants uneasy and will lower the technical benefits of the review.

The technical review process is also fairly formal. The manager schedules the preparation and the review meeting and presents the material to the reviewers. The reviewers are usually expected to examine the material for defects and issues before the review meeting is held.

At the review meeting the chair provides an overview of what is going to happen. The document is leafed through page by page and issues are noted and discussed. Conclusions about what should be changed and what should not should be reached before the end of the meeting.

The author is present, but in contrast to his or her role at a walk-through, the author should stay silent and listen during a review meeting. Clarifying questions may be asked and answered, but the author should not try to “defend” him- or herself.

A report should be written after the review meeting summarizing the findings and the conclusions. In some cases measurements related to the time and defect finding are reported.

If the document is rejected and a new review is to be performed, this must be scheduled by the manager.

If the document needs rework before it can be approved, this will take place after the meeting and is usually done by the original author.

The disadvantages of technical (and management) reviews are few. The outcome depends on the reviewers, but these can be selected carefully to get the best results. If reporting of measurements is not imposed it is difficult to calculate the effectiveness of the technical reviews, but the measurements are not difficult to obtain.

The benefits of technical reviews are even greater than those for informal reviews. Defects are found early and cheaply, information about the quality of the produced objects is gathered, and the participants learn from each other.



### 6.2.4 Management Review

Management review is a review type performed on management documents. This may be:

- ▶ Project-related plans, such as:
  - ▶ Project management plans, including schedules and resources
  - ▶ Quality assurance plans
  - ▶ Configuration management plans
  - ▶ Risk management plans
  - ▶ Contingency plans
- ▶ Plans pertaining to the product, such as:
  - ▶ Safety plans
  - ▶ Installation plans
  - ▶ Maintenance plans
  - ▶ Backup and recovery plans
  - ▶ Disaster plans
- ▶ Reports, such as:
  - ▶ Progress reports
  - ▶ Incident reports, including customer complaints
  - ▶ Technical review reports
  - ▶ Inspection reports
  - ▶ Audit reports

The primary objective is to find defects in the documents under static testing. The secondary objective, however, is usually even more important. This objective is to monitor progress according to the current plan, to assess status, and to make necessary decisions about any actions to take accordingly, including changes in resources, time, and/or scope/quality and updating the plan accordingly. The scope and the quality are usually expressed in terms of requirements to fulfill.

Management reviews are usually planned to take place at certain times in the development life cycle, typically in connection with defined milestones, that is, transfer from one development phase to the next.

There is usually no basis material as such unless a document is reviewed in relation to a document standard or the like. On the other hand, a management review is performed using all appropriate information about the status of the project and the product, like progress and status reports concerning both technical and financial aspects and incident reports.

The roles that should be filled for a management review are the decision maker (the owner of the plan), the leader, reviewers, and a recorder. The reviewers are relevant stakeholders and should include management and tech-

nical staff involved in the execution of the planned activities.

For the management review of a test plan the reviewers should include the test manager, the project manager (higher management), and the people involved in the testing activities.



The total number of participants should be within 3–10 people.

The management review process is also fairly formal. The leader schedules the preparation and the review meeting and presents the plan and any other information to the reviewers. The reviewers are usually expected to be prepared, that is to know the current plan and any deviations from it, before the review meeting is held.



At the review meeting the plan is checked for compliance with other plans and consistency with reality. The performance of management procedures being applied may also be assessed. Conclusions about what should be changed in the plan and what should not should be reached before the end of the meeting.

A report should be written after the review meeting summarizing the action items defined and the issues to be resolved, if any. In some cases measurements related to the time and effectiveness of the review are reported.

The disadvantages of management reviews are few. The outcome depends on the reviewers, but these are usually sufficiently committed. If reporting of measurements is not imposed it is difficult to calculate the effectiveness of the technical reviews, but the measurements are not difficult to obtain.

The benefits of management reviews are many. A plan that is agreed on by all relevant stakeholders has a higher probability of being followed than a plan without such an agreement. The benefits of monitoring and control of progress of an assignment are discussed for testing in Section 3.4.



### 6.2.5 Inspection

Inspection is a formal and well-defined type of static test. The technique was first introduced in 1972 in IBM by Michael Fagan, and since then the inspection process has evolved through use in regular development and experimentation.

Fagan inspections have a number of specific characteristics, which must all be observed before a static testing activity may indeed be called an inspection. The characteristics are:

- ▶ The process to follow must be the formally defined process
- ▶ The roles must be the defined inspection roles
- ▶ Source material (basis documentation) must always be used
- ▶ The inspectors must look for specific kinds of issues
- ▶ Metrics must be defined and collected



- ▶ Process improvement is an integrated part of the process
- ▶ The moderator and the inspectors must be trained

Inspections have two official purposes:

- ▶ Product improvement
- ▶ Process improvement

As with the other static testing types the main objective of inspections is to find defects and thereby contribute to the improvement of the product.

The secondary and almost equally important objective of inspections is contribution to process improvement. This is primarily aimed at improvements to the inspection process itself, but other processes may also benefit from the results of inspections. The process improvement objective is supported by the collection and analysis of measurements for all inspections.

The formal inspection process consists of the activities:

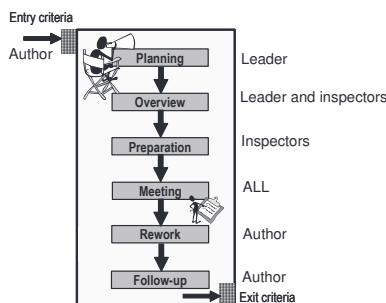


- ▶ Planning
- ▶ Overview
- ▶ Preparation
- ▶ Meeting
- ▶ Rework
- ▶ Follow-up

The roles in play in inspections are:

- ▶ Inspection leader
- ▶ Author
- ▶ Inspectors
- ▶ Moderator (meeting chair)
- ▶ Recorder

The involvement of the roles varies from activity to activity during the course of the inspection process, as shown in the following figure.



### 6.2.5.1 Inspection Leader

The inspection leader is responsible for the performance of the inspection process in a project or an organization. This responsibility includes:

- ▶ Selection of the inspection team, not least of all the inspectors
- ▶ Planning of the inspection
- ▶ Monitoring and controlling the course of the inspection, and taking corrective action as required
- ▶ Verifying if the entry criteria are met
- ▶ Conduction of the overview activity
- ▶ Being an active part of the follow-up activity
- ▶ Verifying if the exit criteria are met and closing the inspection
- ▶ Informing stakeholders of the results
- ▶ Contributing to process improvement of the inspection process and other processes in general



It is quite often seen that the inspection leader also fills the role of moderator, and takes part in the inspection meeting in that capacity.

An inspection leader must be trained in the inspection process, in the metric plan in the organization, in general process improvement, in statistical improvement methods, and in the organization's inspection policy.

### 6.2.5.2 Inspection Entry and Exit Criteria

Entry and exit criteria must always be defined and adhered to for an inspection. The inspection leader carries the overall responsibility for ensuring that the criteria are met, though the author is responsible for making sure they are.

The purpose of the entry criteria is to make sure that time is not wasted by starting the inspection process too early. The document to test must be in a reasonable state, so that any issues can be easily handled and corrected.

Entry criteria may be:

- ▶ Relevant checklists are available
- ▶ Basis documents have exited their inspections with a known and acceptable remaining defect level
- ▶ In 10 minutes of trial examination not more than one major defect is found
- ▶ The document has gone through spell-checking, indexing, and grammar check as appropriate



Exit criteria must be met in order for the document to be approved and the inspection to be officially finished. Exit criteria are therefore defined both for the document and for the inspection process.

**Ex.**

Exit criteria for the document being inspected may be that the entry criteria are still met, that any necessary rework has been performed, and that this rework has been verified by the inspection leader.

For the inspection as such the exit criteria may be that the checking and logging rate were within the prescribed limits and that the estimate for the remaining defects in the document is below an acceptable limit.

### 6.2.5.3 Inspection Planning

The planning activity is concerned with the document to be inspected, the people involved, the metrics to collect, and the schedule and logistics.

The planning must take the nature of the document to be inspected into account. This includes the size and the complexity of the object. Depending on the document the optimal checking rate must be determined. The checking rate is the number of units, typically pages, to be inspected per time unit, typically hours, by the inspectors. Measurements collected for inspections should be used to determine the optimal checking rate for the various types of objects appropriate for the organization.

The size and complexity of the document as well as the result of a risk analysis can indicate if the entire document is to be inspected in one go or if chunking or sampling should be used. Inspection is an intensive static testing technique, and it is usually not possible to make a complete inspection of a document in one single inspection cycle.



Chunking means that the document is divided into series of a few pages (or other units as appropriate) at a time. All relevant parts of the document are checked in a number of preparation-to-follow-up cycles. This technique has the very useful side effect that defects found in one chunk can be corrected in other chunks before these are subjected to the inspection.



Sampling means that only selected part(s) of the document are to be checked. Based on the results of the inspection of the sample, defect densities for the uninspected checked parts can be calculated. Sampling is a cheaper alternative to checking the entire document, and it can be used to provide a firsthand measure of quality, to train the author or other inspection roles, and to gain insight into defect patterns. The underlying idea of sampling is that defects in one part of a document often exist in other parts as well. Having found the defects in the sample, we can look for them in the rest of the object. Less formal static testing types may then be used on the rest of the object to sweep up.

The planning must also deal with the people to be involved in the inspection. It must be decided who is going to fill which roles, and it must be en-

sured that the right people are available and able to participate as needed.

The most important role to fill at this point, where the author is given as the author of the document to inspect and the inspection leader is already at work, is that of inspector or inspectors. Inspectors may be selected according to specific skills, technical or domain knowledge, and/or availability.

Inspectors may have individual inspector roles. The inspector roles are subroles related to the inspector role, and not to be mixed up with the other inspection roles. The concept of inspector roles is discussed below.

The measurements to be collected during inspections should include the following:

- ▶ Sizes
- ▶ Defects found—classified after severity
- ▶ Cost of corrections
- ▶ Time used on the individual activities

The sizes include the total size of the object and possibly size(s) of inspected chunks or samples. It also includes the number of participants, classified by roles.

Cost of corrections includes time  $\times$  rate of those involved in the rework and other correction activities plus any additional expenses.

The time used on the individual activities include planning time, time used per role for the overview, time used per role in preparation, and time used per role in the inspection meeting. This latter time may be collected as time used for logging, for discussion time, and for formalities and other things. Last we need the time used for follow-up.

The planning includes the production of the inspection schedule. This must be based on the overall schedule for static testing and for the project, and the availability of the object, the base document, and the people filling the roles. It is important that the specific preparation time is calculated for each individual preparation to be performed based on the optimal checking rates.

#### 6.2.5.4 Inspection Overview

The overview is the kickoff of the inspection. The overview should include:

- ▶ Group education of inspectors
- ▶ Assignment of inspector roles
- ▶ Distribution of material



The overview is the place where the inspection leader introduces the inspectors to the specific inspection to be done.

The group education of the participants has two goals, namely, education in the inspection process to be used in this particular inspection, and, in the material to be inspected. The plan for the inspection, not least the time assigned for each individual to prepare, should be presented. Information about the current state of inspections in the organization, including relevant statistics and future plans, should be given. This may put this particular inspection in perspective and may be a good motivational factor.

More formal education leading to certification will have to be done as part of the organization's training program apart from the actual inspections.

The introduction to the material to be inspected could be in the form of somebody paraphrasing it, or making a short presentation of such factors as purpose, structure, content, and layout. This may resemble a walk-through but the presentation should not go into the actual contents of the material.

The overview ends with the assignment of specific inspector roles and handouts of object and base material. It is important that the inspection leader makes sure everybody understands his or her role and is happy with it, and that he or she knows what other roles are being covered. The inspector roles are explained below.

An actual overview meeting is not necessary if everybody involved is very experienced. In this case the roles may be assigned individually and the material just sent out to the inspectors.

#### 6.2.5.5 Inspection Preparation

The preparation activity is the heart of the inspection. This is the execution of the testing.

During the preparation each of the inspectors performs the individual checking of the object according to his or her assigned inspector role using the time granted.

Each inspector performs the preparation when and where it suits him or her. As an inspector you should make sure that you can sit undisturbed for the entire preparation time to be used and of course that you can perform the preparation before the inspection meeting is to take place.

The inspectors must follow the standard requirements for inspection preparations, namely:

- ▶ Note starting time
- ▶ Note down or mark issues
- ▶ Focus on majors, at least in the beginning

- ▶ Use the given time (no more, no less) to respect checking rate
- ▶ Count issues

In inspection the issues found during the preparation should be classified as major, minor, or question—marked as M, m, or ?. The definitions of the classifications are:

- ▶ *Major*: The issue seems to concern more than the inspected object and/or will cause further damage.
- ▶ *minor*: The issue is not likely to cause damage to anything beyond the inspected object.
- ▶ ?: The issue needs to be clarified by the author

Note that even though the object is the one being inspected, issues may be found in the base documents as well. These should of course also be noted and reported.

#### 6.2.5.6 Inspector Roles

The inspector, or rather usually the inspectors as most inspections include a number of inspectors, are the people doing the actual checking of the document.

The inspectors must be people who like to find defects. It may sound stupid, but many people are uneasy about finding defects in other people's work and this attitude may jeopardize the effectiveness of the inspection, and make it a very unpleasant experience for the inspectors.

During the preparation the inspectors must keep the checking rate they have been given. They must make sure that they have finished the checking in time for the meeting, or they must give notice of problems to the inspection leader as soon as they occur.

The inspectors must also adhere to the inspector role they may have been assigned.

The concept of inspector roles is based on the facts that we tend to see what we are looking for and that we are usually only able to keep focus on one or two things at the time.

What do you see if you are asked to inspect the flowers in the drawing shown here?

If you are not asked to look for something specific you may count the flowers, note the color of the flowers, count the number of leaves, see if there are more buds than open flowers, maybe note the band and its color, and check if the number of stems corresponds to the number of flowers and leaves.

That is all fine, but what about the faces hidden in the bouquet? Did you see them? If not, look again before you turn to the solution in Appendix 6A.



The inspectors may be given specific inspector roles to make each of them focus on just what he or she is supposed to look for and hence make sure that all important aspects are given first priority by at least one inspector.

Inspector roles may be chosen among the following (ordered alphabetically) or others that may be relevant in the context:



*Checklist:* The inspector must inspect the document in relation to specific or generic checklists. Checklists may summarize rules, previously experienced problems, usability, or any other imaginable aspect of the particular object.

*Documents:* The inspector must inspect the consistency between the document and one or more other documents.

*Focus:* The inspector must inspect the document looking for a specific kind of problem. The focus could be on usability, system implications, financial aspects, readability, grammar, cross-references, domain terms use, or perspective (for video games or the like). This inspector role may be somewhat less objective than the others unless specific criteria have been defined.



*Perspective:* The inspector must inspect the document using a more active approach. The perspective is intended to represent or mimic a specific role of a future user of the object. This could, for example, be a designer, a tester, or a programmer. It is important that the inspector having a perspective role either normally fills this role or understands it fully. The perspective role is less specific than the scenario role.

*Procedural:* The inspector must inspect the document following a specific procedure, for example, reading the document backwards.

*Scenario:* The inspector must inspect the document following a specific work procedure for a specific role. This inspector role is therefore more specific than the perspective role. Scenarios with accompanying questions must be created. This can be a significant task but can on the other hand help avoid duplication of inspection efforts.

*Standard:* The inspector must inspect the document against external standards, for example IEEE, BS, or ISO standards, or internal standards such as company, organizational, business, project, or phase-specific standards.

*Viewpoint:* The inspector must inspect the document as, for example, a user, an analyst, a tester, or a designer. This role is very similar to perspective, but relevant base documents are also used.

No matter which inspector role an inspector is to follow, issues falling under another role may be found and these should of course also be noted and logged during the inspection meeting. So should any issue identified in the base documents.

#### 6.2.5.7 Inspection Meeting

The purpose of the inspection meeting is to log the issues found during the preparation. It may seem unnecessary to have a meeting if all issues are noted down. However, experience shows that about 10–20% more issues will be

found during a meeting. The meeting therefore enhances the effectiveness of the inspection.

If only very few issues are found during preparation or only one or two inspectors participate in the inspection, the meeting may be skipped.

The moderator conducts the meeting. He or she must be neutral with respect to the document being inspected and any base documents used. The moderator should be trained in conducting inspection meetings and know the rules. Being a good moderator requires good training, good people skills, and a good sense of judgment.

The moderator must be able to:

- ▶ Cancel the meeting if anybody is unprepared
- ▶ Be strict and diplomatic
- ▶ Ensure that the logging rate is kept
- ▶ Not allow discussions to evolve
- ▶ Keep issues aimed at the product, not the producer
- ▶ Keep solution-oriented statements to a minimum
- ▶ Make unpopular decisions like evicting someone or stopping the meeting
- ▶ Keep participants with strong technical skills but low social skills from “killing” each other



It may sound as if an inspection meeting is a terrible ordeal, and it may indeed be if care is not taken to make it the productive and inspiring activity it is supposed to be.

The first activity is to collect specific information, such as the present date and time and the names of the participants. The total number of issues found by each of the inspectors must also be reported and registered.

When the formalities are being dealt with the logging can begin. The moderator goes through the inspected document line by line, section by section, or using any other unit of the document that is relevant. For each unit the inspectors report their findings in turn, and the recorder logs the issues.

The reporting should concentrate on majors and questions. Minors can be given to the recorder later, as long as the relevant measurements are collected and registered.

When new issues are encountered during the meeting, these must be recorded as well and they must be marked so that it is possible to see how many new issues an inspection meeting spurred.

A logging form may look like this:

**Ex.**

Issue no.	M, m, o, n	Description	Document	Page/line
1.	M, o	No. scheme is unclear	-	p. 3, 143 (17)
2.	M, o	Testgroup missing no. (CLL)	Systesplan	p. 5-6
3.	M, o	Heading 'User setup' not found in testplan	systesplan	p. 5
4.		Term 'testingfields' meaning unclear	-	p. 3
5.	m, o	Term 'Setup user screen' not appropriate	-	p. 3
6.	m, o	Numbers not ordered	-	p. 3 purpose
7.	m, o	'Validations' meaning unclear	-	p. 3 purpose
8.	M, o	71, 72, 77, 65 is heading not	-	p. 3 purpose

In the second column M and m stand for Major and minor, respectively, as explained above. o stands for old: issue found during preparation, and n stands for new: issue found during the inspection meeting.

The recording must be done in such a way that the author will be able to understand it afterwards. The recorder should be able to keep up with the logging rate. The logging rate should be kept high. As a rule of thumb, one to two issues should be recorded per minute. The moderator should check regularly, at least at half time to see if the rate is kept.

No solutions must be suggested or other discussions started during the recording. This would lower the logging rate significantly and would probably not produce a useful solution anyway. If needed, a solution-finding meeting may be scheduled to take place later.



An inspection meeting should not last longer than one hour without a break, not longer than two hours in total. This is important in order to keep logging rate high and focused. Breaks can, however, be used to smooth feathers, console the author, or keep people in line.

At the end of the inspection meeting a conclusion must be reached as to whether the document is accepted, accepted with rework, or rejected. If the document is rejected it must be sent back to the author for further work and another static test, maybe an inspection, maybe a review, will have to be scheduled.

If rework must be done, then this is the next activity in the inspection process.

### 6.2.5.8 Rework After Inspection

Rework is the activity in the inspection process in which the issues found during the preparation and the meeting are dealt with. The issues concerning the document are resolved by the author or the editor of the document in accordance with the basis documentation.

Issues in other documents outside the authority of the author or editor must be directed to the appropriate people by the use of incident registrations.

### 6.2.5.9 Inspection Follow-Up

The inspection process cannot be finished before the exit criteria have been met. The inspection leader must make sure that the document has been corrected and that this correction has not resulted in too many additional defects.

The inspection leader cannot be responsible for the actual work, so this may require one or more informal or perhaps formal reviews. All issues must be classified as either corrected, no correction done or needed, or issue redirected.

The document should now be ready for others to use and to be placed under configuration management as ready according to the project plan.

The follow-up activity includes collecting the final measurements, and performing measurement analysis. Timing measurements must be collected for rework and follow-up, possibly distributed on individual activities such as edit time, time used to write incident reports, approval time, and analysis time.

Based on the measurements and other experience data, the number of remaining defects can be estimated, as may be the cost saved by finding the defects now rather than later.

Any ideas for process improvements must be registered as appropriate.

### 6.2.5.10 Inspection-Based Process Improvement

The process improvement originating from inspections is based on analysis of the collected measurements. From the first pilot of inspections in an organization measures must be collected to show the value of the inspection, calculating all the time saved by finding defects earlier than normal. It is very important for the success of inspections that everybody involved understands the reason for the rigorously formality and is prepared to adhere to the rules.

Improvement of the inspection process in the organization should be ongoing in order for it to be adapted to the best use. Each organization must establish its own numbers for aspects like:



- ▶ Optimal checking rate
- ▶ Optimal logging rate
- ▶ Optimal pages or participants per inspection

These metrics make it possible to create evidence of the benefits, efficiency, and effectiveness of formal inspections.



The practice of inspections in an organization should only continue if it is cost-effective compared to other test methods.

Information gathered from inspections can also be used in process improvement initiatives beyond the inspection process. Root cause analysis of defects found, both defects in the inspected document and in the basis documents, can lead to processes in need of improvement.



For an inspection it appeared that some of the inspectors had used an old version of one of the base documents. When this was followed up on, it revealed a weak process in the configuration management system.

### 6.2.6 Audit

Audit is by far the most formal static testing technique. Audits are performed by one or more external auditors. Auditors must have a special education and certification to perform official audits.

Audits are usually performed late in the development life cycle, just before release of the product and/or closedown of the project. Smaller audits may be performed in connection with other important milestones.

The primary objective of an audit is to provide an independent evaluation of an activity's compliance to applicable process descriptions, contracts, regulations, and/or standards.

Audits may be either internal or external. Internal audits are performed by auditors from the organization, but external to the project under audit. External audits are performed by auditors entirely external to the organization, usually from some sort of authority organization.



An internal audit may be conducted by certified auditors from the organization's method department to evaluate if CMMI®-compliant processes for system and acceptance testing have been followed correctly.

An external audit may be performed by certified auditors from the FDA (the American Food and Drug Association) to evaluate if the FDA regulations have been followed during the entire development life cycle of a product for use in connection with patient treatment at hospitals, and thereby determine if the product can be allowed into the American market.

Any document, technical as well as managerial, can be part of an audit. In an audit a number of documents, if not all documents, will usually be evalu-

ated and their consistency as well as their collective compliance to the basis material checked.

A lead auditor is responsible for the audit, and he or she also acts as moderator. The lead auditor and any other participating auditors collect evidence of compliance through document examination, interviews, and walk-throughs of documents of special interest.

The audit leader decides when the audit should take place, and the documents will be audited in the state they are in at the time of the audit. Usually a fair warning is given and organizations may rehearse the audit to patch up any obvious defects in time.

The result of an audit is a report in which the findings are summarized in the form of lists of observations, deviations, recommendations, and corrective actions to be taken, as well as a pass/fail statement.

The disadvantages of audits are that they are expensive and the least effective static testing type. On the other hand, audits are usually performed because they are mandatory in some context and therefore serve a different purpose than pure defect finding, such as official approval of the product or process improvement.



### 6.3 Static Testing in the Life Cycle

During the course of the development of a product, appropriate static tests should be planned in the project plan, the quality assurance plan, or the test plan.

Static testing can, and should, start as soon as something is officially written down in the product lifecycle. This may be when the first idea begins to take form or when a contract for development is drawn up.

The planning of static tests to be performed may include static tests of specific documents at specific points in time—typically aligned with milestones planned for the development according to the chosen development model.

Development models are discussed in Section 1.1.1

No matter which development model one chooses for a given project, each of the phases usually ends with a milestone with a specified outcome. Milestones may be named after the documents to be delivered at the milestone and the associated static tests may be named accordingly.



The number, names, and contents of milestone deliveries depend on the individual project and the chosen development model.

An example of milestones with associated deliveries to be reviewed in a project following a small waterfall model is:



- ▶ Closing of contract
- ▶ Closing of requirements specification
- ▶ Closing of architectural design

- ▶ Closing of detailed design
- ▶ Acceptance/qualification of complete product
- ▶ Final delivery/operational readiness

The type of static testing for the milestones as well as the objectives of the individual static tests should be selected according to the type of document, the criticality of the product, and the product risks.

The static testing associated with the contract milestone would typically be a management review, involving management, customers, and technical people, for example, from analysis, development, and testing.

Static testing of requirements specifications and design documents may be combinations of walk-throughs, technical reviews, and/or inspections.

The static testing for acceptance and operational readiness may typically be comprised of management reviews of test reports and/or formal audits.

Management reviews of plans and related progress documentation should also be part of the milestone-related static tests.

The basis documentation to use for these static tests must be documents produced and approved for the previous milestones, as well as those pertaining to standards and regulations. When static testing is performed on a document, defects are often found in the basis documentation as well, and these will change versions as the development progresses.


**Ex.**

The table below shows an example of milestone deliveries and their contents and development over time. The numbers shown for the documents and the product are their respective version numbers. A dash signals that the document/product is not part of the specific delivery.

Milestone delivery	Closing of contract	Closing of requirements	Closing of architectural design	Closing of detailed design	Acceptance of product	Final delivery
Included configuration item						
Contract	1.0	1.1	1.1	1.1	1.2	2.0
Project plan	1.0	2.0	3.0	4.0	4.1	4.1
Test plan	-	1.0	2.0	3.5	4.2	5.3
Requirement specification	-	1.0	1.3	1.4	1.6	1.6
Acceptance test specification	-	1.0	1.3	1.4	1.5	1.5
Architectural design	-	-	1.0	1.2	1.3	1.3
Detailed design	-	-	-	1.0	1.1	1.2
User manual	-	-	-	-	1.0	1.1
Complete product	-	-	-	-	1.0	1.1

## 6.4 Introducing Static Testing

The introduction of static testing in an organization is a process improvement project with all that this entails in terms of organization changes. The principles of process improvement in general are discussed in Section 8.2.1.

Management commitment and support are essential for the implementation to be a success. An implementation process must be described and followed closely to enhance the probability of the introduction of static testing processes and techniques being successful.

An implementation process should at least include the following activities:

- ▶ Make necessary new process descriptions and/or necessary adjustments to existing ones
- ▶ Perform a pilot project
- ▶ Assess the pilot project
- ▶ Produce a rollout strategy
- ▶ Make the rollout happen
- ▶ Follow up on the rollout

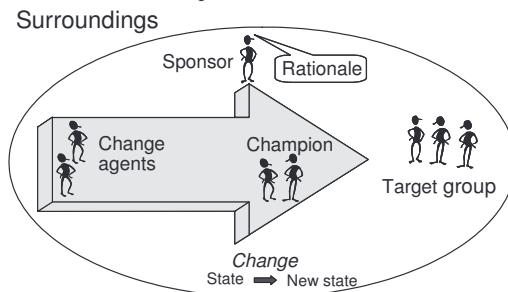


The necessary resources, both in terms of people, time, money, and training must be provided and sustained until the usage of the new procedures is an engraved part of everyday working life.

### 6.4.1 Static Testing Implementation Roles

A number of roles must be in place to make a process improvement, here the introduction and continuous improvement of a static test, a success. The necessary roles are illustrated in the following.

The most important role is the *sponsor*. This must be a senior manager with enough impact both financially and organizationally to support the introduction and improvement all the way.



The sponsor must provide visible support. He or she is the one to approve the project, demand results as the implementation is carried out, and make

necessary decisions on the way. The sponsor must make the rationale for the introduction of the static testing visible to all involved to help overcome the natural resistance to changes that will inevitably occur.

The *target group* consists of the people who are going to be the everyday users of the new processes. They are the ones that need to change their behavior and those most likely to resist the implementation. It is therefore very important that they are involved and heard in both the description of the processes to use and the introduction of them. The target group must also be supported all the way during the introduction.

The support is primarily provided by the *champions*. The champions are a significant force in the introduction. They have day-to-day contact with the target group, and they carry a lot of the responsibility for the success of the introduction.

The champion is the ambassador for the new processes and must be honestly enthusiastic about the benefits. He or she must also be very good at working with people, even if they get negative or discouraged. The champions are best recruited among the people in the target group and must have the trust of the others.

A champion must understand the new processes, and he or she must also understand the issues facing the target group and must be able to combine the two skills to provide the best support to the target group.

The *change agents* are the bridge between the sponsor and the champions. This role has the daily responsibility for the implementation of the processes and is typically a line manager or a manager in a process or a method function manager.

It is the responsibility of the change agent to plan and manage the entire introduction, including the pilot project, and to ensure the completion of the introduction. The change agent could be the same person as the champion or one of the champions.

#### 6.4.2 Static Testing Processes

Before static testing can be introduced, processes for each static testing type must be produced.

A process description may include:

- ▶ Purpose—A description of what must be achieved
- ▶ Entry criteria—What must be in place before we can start
- ▶ A definition of the necessary input – What are we going to work with
- ▶ A list of activities—The procedure, what are we going to do
- ▶ Roles – who are going to perform the activities
- ▶ Methods, techniques, tools—How exactly are we going to perform the activities
- ▶ Templates—What should the output look like

- ▶ Measurements—What metrics are we going to collect for the process
- ▶ A description of the output—What are we expected to produce
- ▶ Exit criteria—What do we need to fulfill before we can say that we have finished

When writing processes for the first time, inspiration can be taken from existing practices in the organization, practices expressed in maturity models, and appropriate process descriptions collected from standards, literature, and available tools.

Processes in general are discussed in Section 2.1.



### 6.4.3 Static Testing Piloting

A pilot project should always be performed for processes before we commit to introducing them across all projects.

There are a number of reasons for performing a small-scale pilot project. First of all we need to get some experience with the processes we have developed. The pilot should enable us to identify necessary adjustments.

A goal for the pilot project is also to verify the business case and ensure that the benefits of the introduction of static testing can in fact be achieved. Finally a pilot can help us refine the estimate for the actual costs and benefits for the introduction.

A pilot should take between three and six months, and be followed closely.

### 6.4.4 Static Testing Rollout

The rollout of the static tests should be based on a successful evaluation of the pilot project. Rollout normally requires the great involvement of all the people filling the different roles, not least the users of the static testing processes: the target group.

First and foremost, the target group must be trained properly and at the right time in the static testing processes they are supposed to follow. Badly timed or inadequate training can ruin an otherwise sound introduction of static testing.

A rollout strategy that suits the nature of the organization must be defined. A “big-bang” rollout, where everybody starts using the new static testing processes at a given point in time, works in some organizations. In other organizations a gradual introduction, where the processes are deployed as the need arises, will work better.

No matter how the rollout is done the most important activity at this point is to support the new users as the rollout takes place. The champions must be prepared to



- ▶ Support the users

until the usage of the static testing processes is a completely integrated part of the daily work.

#### 6.4.5 Psychological Aspects of Static Testing



We must be aware that static testing may end in frustration for both the author and the reviewers when static testing is introduced and even when it is performed on a regular basis.

The author has done his or her best and is perhaps expecting to be praised for the good work during the static testing feedback. This will, however, rarely happen. The reviewers are, as they should be, looking for shortcomings and defects, and even if the work is in fact excellent there will always be something to find, and that should be found and brought forward. The reviewers do not expect the author to be personally offended by the static testing feedback—they are also doing a good job.



Both parties will have to keep in mind that static testing is a necessary and effective activity and that it is there to help increase the quality of the work. There is nothing wrong with making mistakes—everybody does that. And there is nothing wrong with pointing out mistakes, as long as the reviewers keep the static testing objective and the reporting matter-of-fact.



The static testing manager should be attentive to the fact that he or she can enhance the team spirit by using static tests in a constructive way and instill the understanding in all the participants that static tests give a better understanding of the product, the processes, colleagues', and one's own capability, and that static tests help increase the maturity level in the organization.

All participants in static testing, as in all kinds of testing, should learn to give and receive criticism in a constructive way. Chapter 10 discusses this in more detail.

### Questions

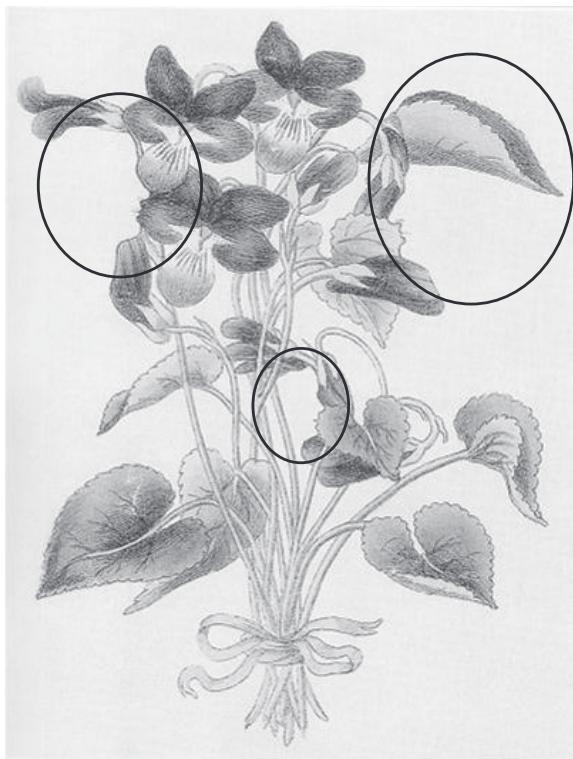
1. When was the concept of static testing first introduced in the software industry?
2. Why is the overall generic term “static testing” used in this book?
3. What are the static testing types discussed in this book?
4. What is usually the object of static testing?
5. How much of the development budget should be reserved for static test activities?

6. How much does the cost of defect correction increase for each phase the defect “survives”?
7. What benefits do static tests have?
8. What are the generic activities involved in static testing?
9. What are the three checking directions to be used in static testing?
10. What are the three possible outcomes of a static test?
11. What are the generic roles in static testing?
12. How should the static testing types to use be selected?
13. Which rule applies when static testing types are mixed?
14. What is special for quality assurance of code?
15. What characterizes an informal review?
16. How should feedback from informal reviews be given and why?
17. What is special about walk-throughs?
18. What can jeopardize the effectiveness of a walk-through?
19. What are technical reviews also called?
20. Who should participate in a technical review?
21. What is the author’s role in a technical review?
22. What is the object of management reviews?
23. What is the main benefit of management reviews?
24. What characterizes an inspection?
25. What are the two official purposes for an inspection?
26. What are the six activities in an inspection?
27. What are the roles involved in inspections?
28. Why are entry criteria defined for inspection?
29. What aspects does the planning deal with?
30. What is chunking, and what is sampling?
31. What metrics could be collected during an inspection?
32. What happens during the overview activity in inspections?
33. What must happen during preparation in inspections?
34. What characterizes a major issue?
35. Why may inspectors be given specific inspector roles?
36. What are the inspector roles mentioned here?
37. What must the moderator be able to do in inspections?
38. What should happen to new issues found during the inspection meeting?
39. How long should an inspection meeting last?
40. What must happen to issues found in basis documents?
41. What are the two main activities in the follow-up?
42. How can inspections contribute to process improvement?
43. Who may perform audits?
44. How can static tests be applied in connection with the development life cycle?

45. What activities should be performed when introducing static tests in an organization?
46. What are the roles to be filled when introducing static tests?
47. What may be included in static testing process descriptions?
48. Why should static testing pilots be conducted?
49. What is most important in connection with rollout of static testing processes?
50. Why may static testing end in frustration?

**Appendix 6A Solution to the Flower Drawing**

The drawing shown in Section 6.2.5 shows the faces of Napoleon, his second wife, Marie Louise, Archduchess of Austria, and his son, Napoleon II, as indicated here.





## 7

**Contents**

- 7.1 Incident Detection
- 7.2 Incident and Defect Life Cycles
- 7.3 Incident Fields
- 7.4 Metrics and Incident Management
- 7.5 Communicating Incidents

## Incident Management

A successful dynamic test detects failures, and a successful static test detects defects. However, if the failures, defects, and other observations made are not managed, the effort is wasted.

The activity of managing what is encountered during testing (and in any other activity for that matter) is called incident management. It is formally an activity in the configuration management process, but emphasized here because it is so strongly connected to testing.

Incident management is about following all incidents from the cradle to the grave and bringing out and using the information embedded in the incidents.

### 7.1 Incident Detection

#### 7.1.1 Incident Definition

This chapter is about incidents. BS 7925-1 defines: *An incident* is every (significant) unplanned event observed during testing, and requiring further investigation.

The standard IEEE 1044 Standards Classification for Software Anomalies, uses the term “anomaly” instead of “incident.” IEEE 1044 defines: “Any condition that deviates from the expected based on requirements specifications, design documents, user documents, standards, etc. or from someone’s perception or experience.” And it adds: “Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.”

This chapter is based on the IEEE 1044 and IEEE 1044.1 standards though the term incident will be used, not the term anomaly.

### 7.1.2 Incident Causes

When a product is being developed, tested, deployed, and maintained, incidents are inevitable. It is human to make mistakes so defects get introduced during development, requirements change over time, and the environment in which the product is deployed can evolve as well. In addition to this, people constantly develop their knowledge about their products and business processes and consequently get new ideas for evolving the product.

Testing is the obvious source of incidents, since the idea of testing is to ferret out things that make us think: “Oops—what was that?” Incidents are detected in static testing, often in the form of defects, and in dynamic testing, typically in the form of failures. An incident could be that the actual result is different from the expected result when a test case is executed. It may also be ideas that arise during testing, both ideas for new test cases and ideas for more or different functionality.

There are many synonyms for an “oops,” for example:



- ▶ Anomaly
- ▶ Bug
- ▶ Deviation
- ▶ Enhancement request
- ▶ Event
- ▶ Failure
- ▶ Problem

The term “incident” is considered more neutral than most of the others commonly used and signifies that what we are dealing with is not necessarily a defect.

Incidents can be raised by all stakeholders within the organization or by customers. It is important that all incidents are handled via a defined path and are processed in a controlled way.

### 7.1.3 Incident Reporting and Tracking



All incidents should be registered or reported. Correct reporting of incidents is important for many reasons.

The more thorough and uniformly incidents are reported, the better the possibilities for making the right decisions in the life cycle of the incident.

Good reports enable exploitation of the information about the product and the processes. The incident reports can be analyzed to find trends in the failures and defects and subsequently to suggest process improvement.



The incident report can be handled on paper or by the use of a tool. The latter is by far the more common these days and modern tools provide facilities for communicating incident reports to those they may concern and to produce statistics very easily.

The incident report must follow the incident through its entire life cycle.

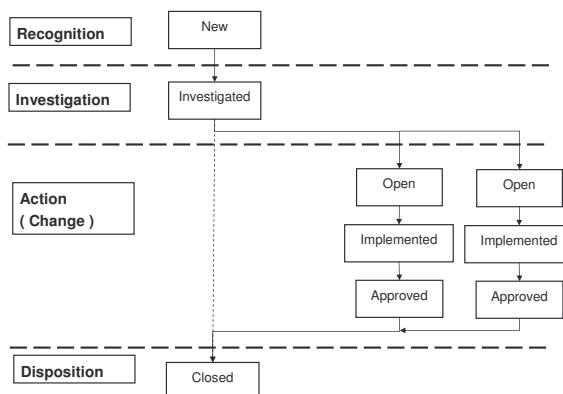
## 7.2 Incident and Defect Life Cycles

When an incident is observed, during testing (either static or dynamic) or during use, and recognized as such, something must happen. But that something is *NOT an immediate change*.

The incident must follow a controlled life cycle.

The life cycle phases defined in IEEE 1044 for an incident are:

- ▶ Recognition
- ▶ Investigation
- ▶ Action
- ▶ Disposition



This is illustrated in this simple life cycle model.

It should be possible to trace incidents through their life cycle and follow their progress. This is done by the use of the incident report that follows the incident and is updated as the incident passes the life cycle phases.

For each of the stages the incident report must contain the following types of information according to IEEE 1044:

- ▶ Supporting data
- ▶ Classification
- ▶ Identified impact



The incident life cycle shown here does not indicate the organizational units involved. The number of organizational units involved can range from one—in the case of incidents found and resolved during official component testing performed in the development unit—to many different organizational units.

Special care must be taken during the life cycles of incidents in the case of outsourced development and/or testing where organizational units belonging to independent companies spread over large geographical (and cultural) distances may be involved.



### 7.2.1 Incident Recognition

Recognition	supporting data
classification	
impact	
Investigation	supporting data
classification	
impact	
Action	supporting data
classification	
impact	
Disposition	supporting data
classification	
	impact

Upon *recognition of an incident* an incident report must be initialized. All incidents must be reported meticulously so that they can be investigated, recreated if needed, and monitored.

The supporting data to provide in the incident report when an incident is first recognized should encompass:

- ▶ Identification of the incident, including unique number, heading, trigger event, proposed fix, if possible, and documentation (e.g., screen dumps)
- ▶ Identification of the environment, including hardware, software, vendor, item in which the incident was seen, and fix description, if any
- ▶ Identification of the people involved, including originator and investigator
- ▶ Related time information, for example, system time, CPU time, and wall time as appropriate

The information for classification should encompass:

- ▶ Project activity: What were you doing when the incident was recognized?
- ▶ Project phase: What phase was the project in when the incident was recognized?
- ▶ Symptom: What did you see when you recognized the incident?

It could also include information about suspected cause, repeatability, and product status.

Impact information should encompass:

- ▶ Severity
- ▶ Project schedule
- ▶ Project cost

The information concerning impact could also include impact on priority customer value, mission safety, project risk, project quality, and society.

IEEE 1044 suggests standard values for the classification and impact categories. These are discussed in Section 7.3.

An example of how an incident report template may look is given below. Only the top part and the part concerning the recognition information part are shown here—the other parts will follow further below in the appropriate sections.

Note that is a real-life example; the words used are not necessarily IEEE 1044-compliant.

<b>Incident Registration Form</b>	
<b>Number</b>	
Short title	
Software product	
Version (n.m)	
<b>Status = Created</b>	
Registration created by	Date & time
Anomaly observed by	Date & time
Comprehensive description	Include references to attachments, if any.
Observed during	Walk-through / Review / Inspection / Code & Build / Test / Use
Observed in	Requirement / Design / Implementation / Test / Operation
Symptom	Operating system crash / Program hang-up / Program crash / Input / Output / Total product failure / System error / Other:
User severity	Urgent / High / Medium / Low / None

The incident report must now be given over to the right people. *For the time being, the incident is now out of the hands of the testers.*



## 7.2.2 Incident Investigation

The investigation is performed based on the information provided in the incident report. Formally this is not testing but configuration management.

The investigation should be done by an authority appointed to do just that. This authority is normally called a CCB (Change Control Board or Configuration Control Board). The CCB must be formed by people with the right insight and the right power to be able to decide what is going to happen to an incident. This includes technical, economical, and possibly political insight and power. The composition of the CCB depends on the extent of the possible impact of the incident—the wider the impact the more formal the CCB.

During component testing the CCB may be the developer or the developer and the project manager.

For high-impact incidents found in system testing the CCB may include the project manager, the product manager, marketing, and the customer.



The investigation is about finding out what is wrong, if anything, and what should happen next. Many things could be wrong, for example, in the context of testing; it could be:



Recognition	A wrong wording, caught during a review of a document
supporting data	
classification	
impact	
Investigation	A coding defect found during a walk-through of a piece of source code
supporting data	
classification	
impact	
Action	A failure found in the integration test
supporting data	
classification	
impact	
Disposition	A wish to expand or enhance the finished product, arising when the product is in acceptance testing
supporting data	
classification	
impact	

- ▶ A wrong wording, caught during a review of a document
- ▶ A coding defect found during a walk-through of a piece of source code
- ▶ A failure found in the integration test
- ▶ A wish to expand or enhance the finished product, arising when the product is in acceptance testing
- ▶ A change required in the code because of the upgrade to a new version of the middleware supporting the system (e.g. a new version of Microsoft Access, which in certain places is not backward-compatible)

If something is indeed wrong the investigation must try to find out what the impact is and what the cost of making the necessary correction(s) is. It must also be considered what the cost of not making the correction(s) is.

It is not always a simple matter to perform such an analysis, but it must be done before an informed decision about what to do can be made.



*Possible actions may be:*

- ▶ Nothing—No failure after all or the failure is too insignificant
- ▶ Nothing right now—Changes are postponed
- ▶ Changes must be implemented immediately where necessary

The supporting data for the other life cycle phases include primarily

- ▶ Identification of the people involved in the investigation, at least those responsible for any decisions made
- ▶ Related time information

The information for classification should encompass:

- ▶ Actual cause—Where have we pinpointed the incident to come from a high level?
- ▶ Source—In which work product(s) or product component(s) must changes be made?
- ▶ Type—What type of incident are we dealing with?

The impact information for this phase is the same as for the recognition phase: severity, project schedule, and project cost as mandatory.

IEEE 1044 suggests standard values for the classification and impact categories. These are discussed in Section 7.3.

The investigation part of the incident report is shown here.



<b>Status = Investigated</b>	
Forwarded by	Date & time
Investigated by	Date & time
Actual cause	Software / Data / Test system / Platform / User / Unknown
Source	Specification / Code / Database / Manual / Plan
Overall problem type	Logical / Computation / Interface / Timing / Data handling / Data / Documentation / Document quality / Enhancement / Failure caused by previous fix / Performance / Other :
Affected Cls	
State type from list above	
Estimated correction effort	
Estimated confirmation test and regression test effort	
Schedule impact	High / Medium / Low / None
Cost impact	High / Medium / Low / None
Evaluator severity	Urgent / High / Medium / Low / None

**Ex.**

### 7.2.3 Incident Action

If any action is to be taken it will be in the form of one or more changes, since one incident may give raise to changes in more places.

It could well be that an incident encountered during system testing requires a change in the requirement specification, in the design, in the code, in the user manual, and maybe even in the project plan.

**Ex.**

Specific change requests should be produced for all the objects to be changed. This makes it easier to followup on the progress of a change through its life cycle: open, implemented, and approved.

The information for classification should encompass:

- ▶ Resolution—When are we going to do something about it?

This part of the incident report from our example is shown on the next page.

Status = Action	
Forwarded for decision by	Date & time
CCB decided	Date & time
CCB's decision	Immediate / Eventual change / Deferred / No fix
Observer informed by	Date
Change request(s) opened by	Ref.: Date & time
All change requests accepted closed by	Date & time
Comprehensive solution description, if applicable	Include references to attachments, if any.
Total actual change effort	
Total actual test effort	
Solution complete CCB signature	Date & time



The testing comes into the incident handling at the time of approval. Retesting must be done to ensure that corrections have been made correctly, and regression testing must be performed to ensure that the correction has had no adverse effects on the areas that were working before the correction.

#### 7.2.4 Incident Disposition

If action has been taken *disposition* can only happen once ALL the change requests have been approved. In this case the incident report is closed with information about how the corrections have been implemented and finally approved.

The information for classification should encompass:

- ▶ Disposition—Why was the incident closed?

The last bit of the example incident report is shown here.

Status = Closed	
Close condition	Closed / Deferred / Merged / Referred Reference:

<b>Status = Closed</b>	
Conditions – if applicable	All new configuration items correctly identified
	All new configuration items properly stored
	All stakeholders informed of new configuration items
Remarks	
Incident observer informed by	Date & time
CCB Signature	Date & time



### 7.3 Incident Fields

It can be difficult to get an overview of a large number of incidents and to be able to see patterns in them. To be able to extract some of the interesting information about the incidents, it is necessary to be a bit systematic about the data being gathered.

A classification scheme provides a standard terminology and facilitates communication and information exploitation within or between projects and organizations.

IEEE 1044 defines a classification scheme for each of the life cycle phases for the incident, namely recognition, investigation, action, and disposition, as indicated in the earlier sections.

The defined classification has a hierarchical structure. This means that there exist a varying number of layers of possible values for each category. In the standard each of the classification values has a code in the form of a unique identification determining its place in the life cycle and the hierarchy. This numbering scheme also means that organizations can use each other's classifications for incident reports even if the actual wording of the values is different, for example due to usage of the local language.

You can use the IEEE 1044 classification scheme as an inspiration to get more structure into the incident reporting. If for some reason you need to be IEEE 1044-compliant (for example, for safety critical software) you need to be aware of the fact that some of the categories are mandatory and some are optional.

An extract of the IEEE 1044 classification scheme is shown in Appendix 7A.

Do not learn the classification schemes by heart, but take them to heart and use them as inspiration for an efficient and useful way of reporting incidents.



### 7.4 Metrics and Incident Management

There is no reason to collect information about incidents if it is not going to be used for anything.

On the other hand, information that can be extracted from incident reports is essential for a number of people in the organization, including test management, project management, project participants, process improvement people, and organizational management.



If you are involved in the definition of incident reports, then ask these people what they need to know—and inspire them, if they do not yet have any wishes!

The primary areas for which incident report information can be used are:



- ▶ Estimation and progress
- ▶ Incident distribution
- ▶ Effectiveness of quality assurance activities
- ▶ Ideas for process improvement

Section 4.3 discusses metrics and measurements in detail.

Direct measurements may be interesting, but they get even more interesting when we use them for calculation of more complex measurements. Like for other areas of life, measurements work best if we get our relationships right.

**Ex.**

It does not say much if we are told that testing has found 543 faults. But if we know that we have estimated that we would find approximately 200, we have gotten some food for thought.

Some of the direct measurements we can extract from incident reports at any given time are:

- ▶ Total number of incidents
- ▶ The number of open incidents
- ▶ The number of closed incidents
- ▶ The time it took to close each incident report
- ▶ Which changes have been made since the last release

The incidents can be counted for specific classifications, and this is where life gets so much easier if a defined classification scheme has been used.

We can count the number of:

- ▶ Incidents found during review of the requirements
- ▶ Incidents found during component testing
- ▶ Incidents where the source was the specification
- ▶ Incidents where the type was a data problem

just to mention a few of the possibilities.

We can also get associated time information from the incident reports and use this in connection with some of the above measurements.

For *estimation and progress* purposes we can compare the actual time it took to close an incident to our estimate and get wiser and better at estimating next time. We can also look at the development in open and closed incidents over time and use that to estimate when the testing can be stopped.

For *incident distribution* we can determine how incidents are distributed over the components and areas in the design. This helps us to identify the more fault-prone, and hence high-risk, areas. We can also determine incident distribution in relation to work product characteristics, such as size, complexity, or technology; or we can determine distribution in relation to development activities, severity, or type.

For information about the effectiveness of quality assurance activities we can calculate the Defect Detection Percentage of various quality assurance activities as time goes by.

*The DDP is the percentage of faults in an object found in a specific quality assurance activity.* The DDP falls over time as more and more faults are detected. The DDP is usually given for a specific activity with an associated time frame (for example, DDP for system test after three months' use).



In the component test 75 faults are found. The DDP of component test is 100% at the end of the component test activity. In the system test another 25 faults that could have been found in the component test are found. The DDP of the component test after the system test is therefore only 75%.



The DDP may fall even further if more component test-related faults are reported from the customer.

The information extracted from incident reports may be used to analyze the entire course of the development and identify ideas for process improvement. Process improvement is, at least at the higher levels, concerned with defect prevention. We can analyze the information to detect trends and tendencies in the incidents, and identify ways to improve the processes to avoid making the same errors again and again—and to get a higher detection rate for those we make anyway.



There is more about monitoring and control of the testing process in Section 4.3.

## 7.5 Communicating Incidents

Careful incident reporting in written incident reports using an incident management system facilitates objective communication about incidents and defects. However, this cannot and should not eliminate verbal communication.



Communication about incidents is difficult. The first thing both testers and developers must be aware of is the danger of “them and us.” We (the testers) may have a tendency to think that we are better than them: “We” are good, conscientious, and right while “they” are careless and evil. They (analysts and developers) may have a tendency to think that they are better than we are: “They” are smart, fast, and pragmatic while “we” are stupid, unknowing, and sticklers for the letter of the law.

It is tempting to fall into the trap of irritation and blame. The first and most important thing for testers and developers is to keep in mind that developers and others do not make defects on purpose to annoy us, or to tease us, or even to keep us busy. The next and equally important thing is that testers and others do not report incidents to gloat and punish but as a their contribution to a high-quality product.

**Ex.**

~~Examples of what NOT to say:~~

~~Tester: “Now you have delivered some .... again – are you never going to get any better?!”~~

~~Developer: “It works on my machine – it must be your setup (or you) that is wrong!”~~



Have mutual respect! Managers and employees alike must work on the spirit of: We are in this together.

Another aspect of incident communication is concerned with what should be corrected and when. This is where it is important to establish CCBs before things get hot and to establish them in such a way that their decisions are trustworthy and respected.



If proper CCBs are not established the decisions about what should happen to incidents and their relative prioritization can be arbitrary and counter-productive, or worse.

## Questions

1. To which process area does incident management belong?
2. What is an incident?
3. What are some of the other names for an incident?
4. What are the four phases in the incident life cycle?
5. What supporting data should be registered in the first incident life cycle phase?
6. In which of the phases in the incident life cycle are the testers not directly involved?
7. What are the possible actions for an incident?
8. What is the prerequisite for closing an incident after correction?

9. What types of information must be given for each phase in the incident life cycle?
10. Why is it a good idea to use a classification scheme?
11. What can we use incident information for?
12. What is DDP, and how is it calculated?
13. What are the two most important things to keep in mind when communicating about incidents?
14. Why is a CCB necessary?

## Appendix 7A Standard Anomaly Classification

IEEE 1044-1993 Standard Classification for Software Anomalies (Extract)

<i>Recognition</i>	<i>Investigation</i>	<i>Impact</i>
<i>Supporting data</i>	<i>Supporting data</i>	
<i>Classification</i>	<i>Classification</i>	
Project activity	Actual cause	
Analysis	Product	
Review	Test system	
Audit	Platform	
Inspection	Outside vendor	
Code/compile/assemble	User	
Testing	Unknown	
Validation	Source	
Support/operational	Specification	
Walk-through	Code	
Project phase	Database	
Requirements	Manual and guides	
Design	Plans and procedures	
Implementation	Reports	
Test	Standards/policies	
Operation and maintenance	Type	
Retirement	Logical problem	
Suspected cause	Computation problem	
Repeatability	Interface/timing problem	
Symptom	Data handling problem	
Operating system crash	Data problem	
Program hang-up	Documentation problem	
Program crash	Document quality problem	
Input problem	Enhancement	
Output problem	Failure caused by fix	
Failed performance	Performance problem	
Perceived total failure	Interoperability	
System error message	Standards conformance	
Other	Other problem	
Product status		
<i>Impact</i>		



<i>Action</i>	<i>Impact</i>
<i>Supporting data</i>	<i>Impact (applicable to all)</i>
<i>Classification</i>	<i>Severity</i>
Resolution	Urgent
Immediate	High
Eventual (next release)	Medium
Deferred (future release)	Low
No fix	None
Corrective action	Priority
<i>Impact</i>	Customer value
<i>Disposition</i>	Mission safety
<i>Supporting data</i>	Project schedule
<i>Classification</i>	High / Medium / Low / None
Disposition	Project cost
Closed	High / Medium / Low / None
Deferred	Project risk
Merged with another	Project quality
Referred to another project	Societal



Note that only the values for the mandatory categories and only one layer of values are shown in the extract of IEEE 1044 above.

Also note that the impact classification is identical for each of the phases in the incident life cycle. It must be done by different people for each phase. In each phase the classification must reflect what the impact is estimated to be at that point in time based on the available information.

When you estimate the impact it is important to remember that the nature of the fault does not necessarily tell anything about the failure that may follow.



A spelling mistake may seem like an innocent fault, but on the home page of a company selling translation and writing services it can be perceived as a major failure.

A fault that looks really nasty, like a possible pointer list overflow, may never actually result in a failure, because there is no way the list can get filled up during the use of the system.

## Appendix 7B Change Control Process

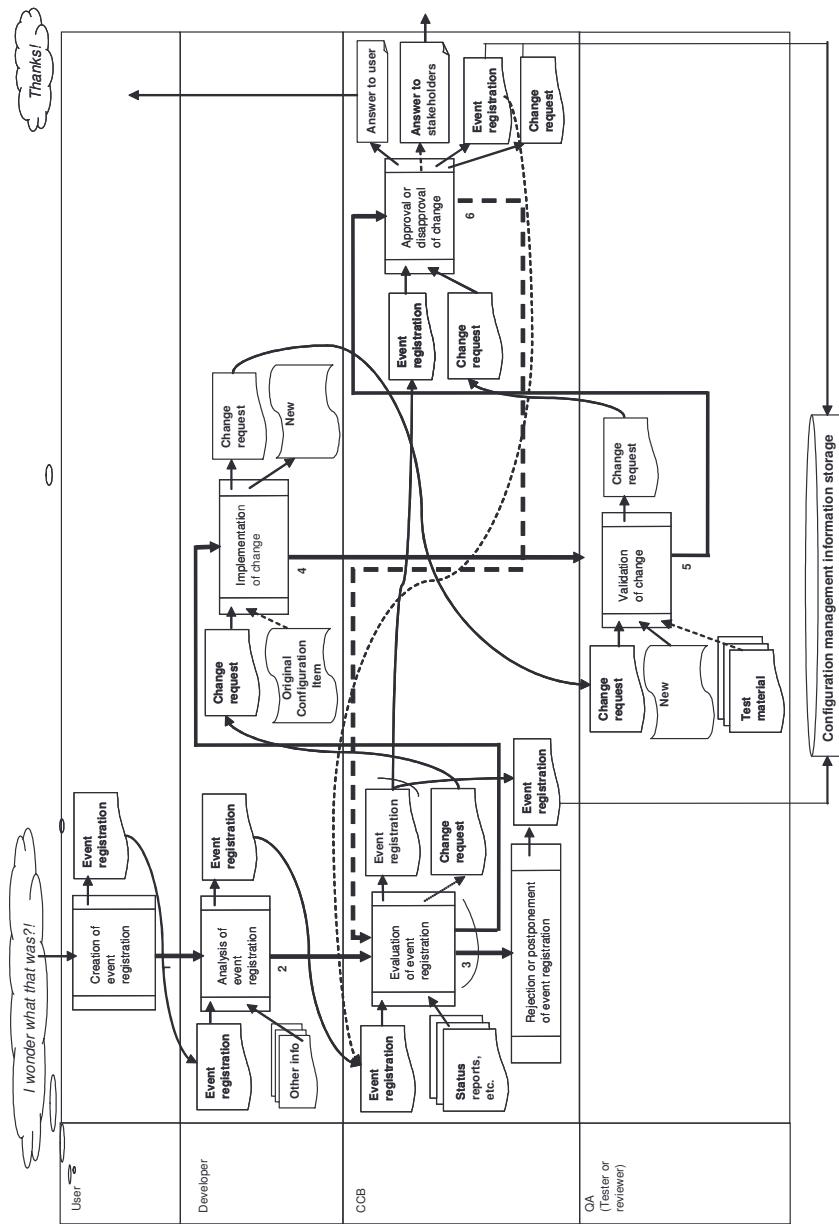
This appendix shows an example of a process diagram for a change control process.

A number of processes are depicted in the diagram as a box with input and output sections (e.g., “Evaluation of event registration”). All these processes will have to be defined, preferably described.

The thick lines illustrate the process flow, and the thin lines illustrate the information flow.

An arch across two lines is used to illustrate “either/or.” A dashed line illustrates “maybe.”

The column to the left holds the name of the role holding the responsibility for the processes shown in the right column.



## Standards and Test Improvement Process

There is never a reason to reinvent the wheel—not even in testing. Many people have worked with testing and related areas for a long time and a lot of their experience is documented and available in standards.

The word “standard” may have a gray and dusty ring to it, but there is nonetheless, a wealth of knowledge in it. All standards have been written and reviewed by a large number of very experienced professionals, and they are a great source of information and ideas for getting started and for getting better.

Process improvement is a discipline that—quite rightly—is getting more and more attention. The demands on good solid processes are growing as the complexity and the demands on the software products and products with software inside are growing.

Process improvement applies to both the entire software development process and detailed processes, such as the testing process. It is usually based on a maturity model. The most widely used maturity models for software development in general are Capability Maturity Model® (CMMI®) and ISO/IEC 15504 (SPICE).

The two most prominent models specifically for the testing area of development are the Testing Maturity Model (TMM) and TPI®. Two other much used models are CTP and STEP.

### Contents

- 8.1 Standards
- 8.2 Test Improvement Process

## 8.1 Standards

### 8.1.1 Standards in General



The word “standard” means a usage or practice that is generally accepted, according to *Collins Pocket English Dictionary*. Standards document experience gained by many people over a long time.

It is a long and hard job to create a standard. The right people have to be found, they have to be able to meet, and—more importantly—to agree enough on the various subjects to determine what “the standard” is. There is usually also a long period of time set aside for hearings and reviews of the material before a standard may indeed be approved as a standard. In some cases this process can take years.



This is part of the reason why standards don’t change very often. They do change when there is a real need because practices, opinions, and experience have changed, and it is important when working with standards to take care that the right version is used and referenced as applicable—versions are often indicated by the year of the issue of the standard.

A standard is not an expression of a scientific truth, but something made by humans. This is why we have a number of standards on the same subjects, for example, testing. And this is why standards sometimes are both internally inconsistent and inconsistent with each other. Standards also to some extent disagree; and it can be difficult to determine which, if any, is most “correct.” Despite this, standards can be a great help.

Standards come from many sources, for example:

- ▶ International standards
- ▶ National standards
- ▶ Domain-specific standards

Standards may also be specific to a specific organization, so-called in-house standards. Such standards are also very useful as guidelines for work to be done.

*Test-related standards fall in three categories*, depending on the type of information they provide.



- ▶ Quality assurance standards—Telling you that *you shall test*. An example of these standards is the ISO 9001:2001 quality management system design.
- ▶ Industry-specific—Telling you that *you shall test this much*. Such standards exist, for example, for aviation, railways, fire alarms, electronic products, vehicles, nuclear plans, and medical devices.
- ▶ Testing standards—Telling you *how to actually test*.

Sometimes we are obliged to follow one or more specific standards because of the nature of the product we are developing or because of demands from customers or management.

Even if that is not the case, we as testers should be aware of which standards are relevant for us and from where we can get guidelines and inspiration.

Remember though, that whatever you do, don't do it because a standard says so. Do it because it serves your business.



### 8.1.2 International Standards

The two most prominent sources of internationally recognized standards are:

- ▶ ISO, International Organization for Standardization
- ▶ IEEE, Institute of Electrical and Electronics Engineers

ISO and IEEE recognize each other, and some standards are common to the two organizations. These standards have the identical numbers in the ISO and, respectively, IEEE series.

#### 8.1.2.1 ISO Standards

ISO is a network of the national standards institutes of 157 countries, on the basis of one member per country, with a central secretariat in Geneva, Switzerland.

The name ISO is derived from the Greek *isos*, meaning “equal,” in order for the name to always be ISO whatever the national language is.

ISO is a nongovernmental organization, but occupies a special position between the public and private sectors.

International standardization began in the electrotechnical field: The International Electrotechnical Commission (IEC) was established in 1906. In 1946, delegates from 25 countries met in London and decided to create a new international organization, of which the object would be “to facilitate the international coordination and unification of industrial standards.” The new organization, ISO, officially began operations on February 23, 1947.

ISO has developed hundreds, if not thousands, of standards. Those of most interest to testers may be:

- ▶ ISO 9000:2005—Quality management systems
- ▶ ISO 9126—Product quality (four different parts from four different years)
- ▶ ISO 12207—Information technology, software life cycle processes



[www.iso.org](http://www.iso.org)



### 8.1.2.2 IEEE Standards

The IEEE is a nonprofit organization based in the United States. The association has more than 370,000 members in over 160 countries.

The full name of the IEEE is the Institute of Electrical and Electronics Engineers, although it is referred to by the letters I-E-E-E and pronounced Eye-triple-E.

The IEEE formed in 1963 with the merger of the AIEE (American Institute of Electrical Engineers, formed in 1884), and the IRE (Institute of Radio Engineers, formed in 1912).

IEEE has issued over 900 active IEEE standards and more than 400 in development on areas ranging from aerospace systems, computers and telecommunications to biomedical engineering, electric power and consumer electronics among others. Those of most interest for testers may be:

- ▶ IEEE 610:1991—Standard computer dictionary
- ▶ IEEE 829:1998—Standard for software test documentation
- ▶ IEEE 1028:1997—Standard for software review and audit
- ▶ IEEE 1044:1995—Guideline to classification of anomalies
- ▶ IEEE/ISO 12207:1995—Standard for software life cycle processes

### 8.1.3 National Standards

Many countries have their own standardization organization, like British Standard (BS) in the United Kingdom, an Dansk Standard (DS) in Denmark. These organizations issue their own local standards, and they may also recognize some of the international standards.

An example of a national standard of interest for testers is:

- ▶ BS 7925-2:1998—Software testing, software component testing



### 8.1.4 Domain-Specific Standards

Many regulatory standards mandating the application of particular testing techniques exist. Some of them (but probably not all) are listed here:

- ▶ IEC/CEI 61508: Functional safety of electrical/electronic/programmable safety-related systems
- ▶ DO-178-B: Software considerations in airborne systems and equipment certification
- ▶ pr EN 50128: Software for railway control and protection systems
- ▶ Def Stan 00-55: Requirements for safety-related software in defense equipment
- ▶ IEC 880: Software for computers in the safety systems of nuclear power stations

- ▶ MISRA: Development guidelines for vehicle-based software
- ▶ FDA: American Food and Drug Association (Pharmaceutical standards)
- ▶ ECSS: European Cooperation on Space Standardization

One of the difficulties with the above standards is that they are not directly aimed at software. Each of them has its origin in a specific field, into which software has only relatively recently penetrated. It would be nice if we had one software-specific standard dealing with the aspects of how much and how to test the software depending on risk analysis. But since that is not the case we will have to make do with the existing standards.

If we are not making software for a discipline dealt with in a specific standard we can always use IEC 61508; this is the most generic of the standards listed above.



## 8.2 Test Improvement Process

Maturity is as important for software development as it is for people. When we are immature we can easily find ourselves in a situation where we lose control and are unable to solve the problems—problems we might even have created ourselves.

The demands on the software industry are growing as pervasive software thunders ahead. More and more products include software, and both embedded software and pure software products are becoming more and more complex. The potential number of faults in the software is hence increasing and so is the cost of finding and removing them—not least keeping in mind that the cost of fault correction increases by a factor 10 for each phase the faults “survive” in the work products.

The solution to the growing demands is more professional software development with focus on the entire product and hence the entire development process. Software development needs to be able to stay in control, foresee problems, and prevent them or mitigate them in a mature way. Software development needs to grow up, improve, and thereby become a mature industry—and so does testing.

Process improvement is based on the understanding that software development is a process and that processes can be managed, measured, and continuously improved.

An important assumption is *that the quality of the software produced using a specific process is closely related to the quality of the process*.

This does not mean that it is impossible to produce excellent software using a useless procedure—or indeed the other way around—but the probability of producing good software rises significantly with the quality of the process.



The urge for improvement can come from many places both outside and inside the organization, and both from below and above.

- ▶ Customers or suppliers may push or even demand proof of maturity and ongoing process improvement directly. More indirectly they may express requirements in terms of quality criteria and time-to-market, whose fulfillment requires a certain maturity in the organization.
- ▶ Within the organization managers are pressed to obey constraints and to provide growth in the organization.
- ▶ Finally employees may well be fed up with constant firefighting and impossible deadlines requiring them to work overtime and cut corners.

**Ex.**

Military organizations usually require their suppliers to be at minimum CMMI® level 3.

### 8.2.1 Process Improvement Principles

Process improvement is hard work. More than anything the software quality is dependent on the abilities of the people working in the organization—both the individuals and the teams. People write the processes and the methods and techniques to be used. People follow the processes and use the methods and techniques.

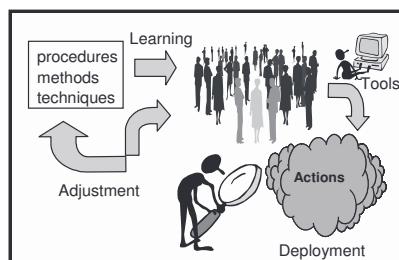


Everything needs to fit together to be efficient and effective.

There are a number of described approaches to process improvement, including:

- ▶ Plan–Do–Check–Act
- ▶ IDEAL: Initialization, Diagnosis, Establishing, Acting, Learning (from SEI)
- ▶ IMPROVE: Initiate, Measure, Prioritize and Plan, Define and Re-define, Operate, Validate, and Evolve (from the ISTQB syllabus)

All these approaches are cyclic as continuous process improvement must be. The following figure illustrates the process improvement process, after the initial phase where the decision to improve is taken, the business objectives and goals defined, the stakeholders identified, and the improvement model chosen.



Organizations performing process improvement must:

- ▶ Produce process descriptions including methods and techniques
- ▶ Introduce the processes to the people working in the organization and provide adequate training
- ▶ Request and support deployment of the processes as appropriate
- ▶ Determine the maturity of the processes and their deployment
- ▶ Use this information to adjust the processes and/or introduce them again to the people
- ▶ Repeat ad libitum



Tools may be used to support the activities, but tools can not provide process improvement by themselves.

A process improvement project must be run like any other project. Activities must be prioritized—both in the short term and in the long run. Activities must be planned with defined goals, activities, resources, time, and budget. The responsible person must follow up on the progress. And last but not least: It is important to report on the successes.

Process improvement is not easy. Many organizations fail their first attempt and that impedes any following attempts. Research has been made into what makes organizations succeed or fail.

This research shows that the *organizations most likely to succeed are those where the software process improvement is incorporated in the organization's visions and strategy at the highest level*.



It is important that top management understand that process improvement is organizational development and that it involves change management. Process improvement is first and foremost about people.

It is hard work that requires continuous focus, but it is also very rewarding work. No organizations I have met have ever wished themselves back to the “bad old days” at level 1 or 2 once they had reached level 3.

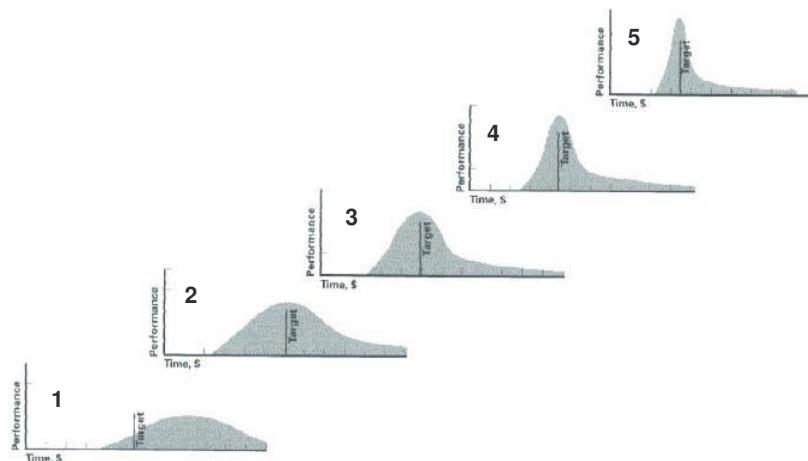
### 8.2.1.1 A Few Process Improvement Results

More and more assessments are being performed all over the world, and companies of all sizes embark on structured process improvement. What most of these companies are asking for before starting a process improvement project is the possible return of investment.

Return on investment (ROI) can be expressed in a number of ways; the most common is of course in economical terms, such as cost and savings, but measures like defects per 1,000 lines of code and employee satisfaction can also be used. The greatest impediment regarding measuring ROI is lack of initial measures. Amazingly many organizations have no clue how much their work costs them.

SEI has collected data from more than 500 organizations using CMM® as the basis for their process improvement. A few of these results are presented here as appetizers.

The following graph shows a typical development in the average and distribution of the performance of a process for organizations moving from maturity level 1 to 5.



At level 1 the average project is far over the target for the estimated time of completion, and the dispersion is high.

The dispersion decreases as the maturity increases, and the average project actually hits the target. As the maturity increases the target is also moved to the left. *Mature organizations perform in a more predictable way, and they perform better than more immature organizations.*

The table below shows the employees' perception of six aspects of the performance of their organization as the organizations moved from CMM® level 1 to CMM® level 3.

**138 individuals**

	CMM® 1	CMM® 2	CMM® 3
Meet schedule	40	55	80
Meet budget	40	55	65
Product quality	75	90	100
Staff productivity	55	70	85
Customer satisfaction	75	70	100
Staff morale	25	50	60

One hundred and thirty-eight people were asked to state their opinion of the aspects on a five-step scale ranging from “nonexistent” to “excellent.” The table shows how many percentages answered one of the two highest possibilities.

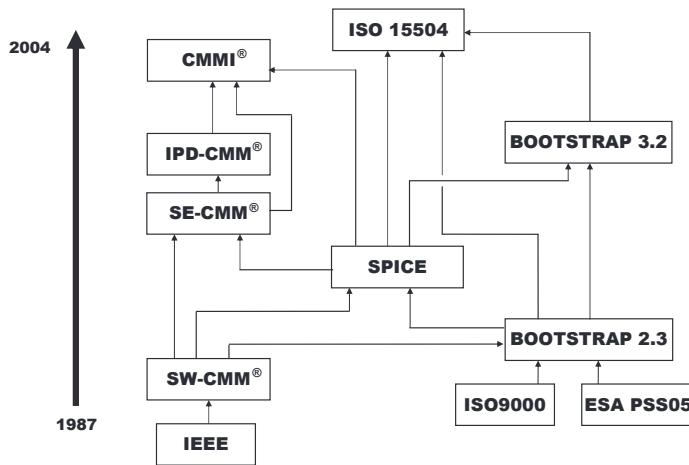
The percentages of employees answering “good” or “excellent” increases with the maturity level.

One small exception is “customer satisfaction” where the high-answering percentage drops to level 2. This could well be because the organizations have started to answer “no” to impossible tasks up-front and maybe also because they are starting to ask the customers for such information as more precise requirements.

### 8.2.2 Process Maturity Models in General

One of the more reliable ways to determine the actual maturity of a software-producing organization is by using a certified software process assessment toolset. An assessment toolset consists of two basic parts: a model and a method. The model is a description of the domain that shall be assessed, and the method describes how to perform the assessment in a verifiable and valid way. The model usually also works as a map guiding organizations towards higher maturity levels.

Many assessment toolsets have been produced during the last two decades, but the most important are CMM<sup>®</sup>, BOOTSTRAP, CMMI<sup>®</sup>, and ISO 15504. These have gotten much inspiration from one another as the “family tree” here shows. BOOTSTRAP is a European toolset now overtaken by ISO 15504.

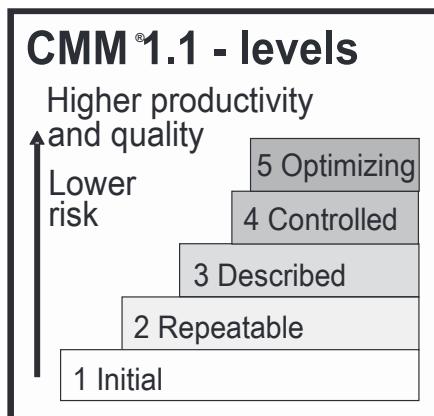


Maybe—one hopes—they will all come together in one standard sometime in the future.

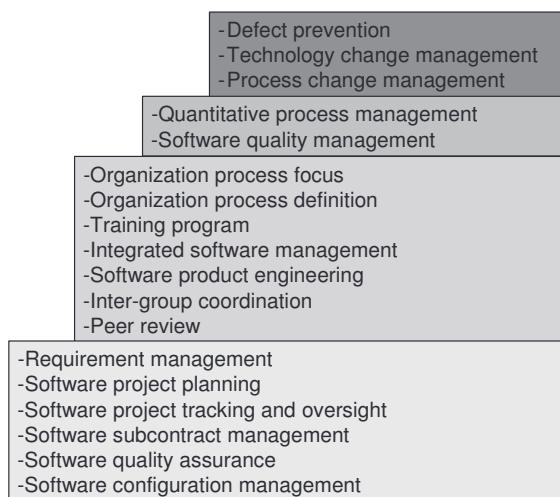
### 8.2.2.1 CMM®

The CMM® was the first assessment toolset, officially released in 1991.

The CMM® model defines five maturity levels ranging from 1 to 5. Level 1 is the lowest; work in an organization at level 1 is adhoc or chaotic. Level 5 is the highest level; organizations at level 5 are continuously optimizing their processes to make the best fit between the business, the people, and the processes.



The CMM® is staged. This means that each maturity level has a number of associated key process areas (KPAs). CMM® KPAs distributed on maturity levels are shown here:



From a testing point of view the CMM® is not adequate. The concept of testing maturity is not addressed and the model does not include testing practices as a process improvement area. Testing issues are addressed in some of the key process areas, but not in a satisfactory manner.

CMM® is still widely used, but CMMI® is moving in fast.

### 8.2.2.2 CMMI®

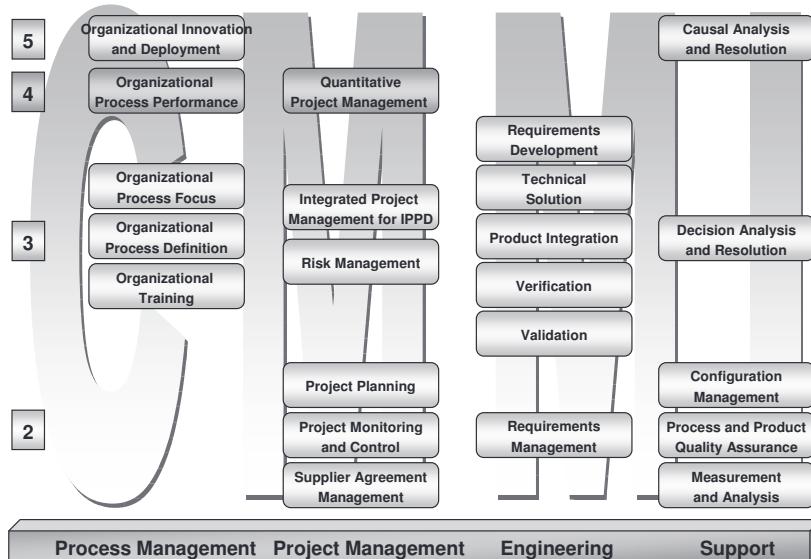
CMMI® Version 1.02 was published in 2000, and Version 1.2 was published in 2006. CMMI® is developed and supported by the Software Engineering Institute, like CMM®.

CMMI® has two representations, namely staged and continuous. Guidelines are offered on how to choose between the models and how to tailor the chosen model to specific needs.

The *staged* CMMI® representation is similar to CMM® V. 1.1. Maturity levels in the staged representation apply to an organization's overall process capability and organizational maturity. The result of a staged assessment is one number: the maturity level.

The capability levels in the *continuous* representation can be reached for each process area individually. There are six capability levels, numbered 0 through 5. The result of a continuous assessment is a profile showing the capability level for each process area.

CMMI® operates with process areas. These are divided into four main groups, namely process management, project management, engineering, and support.



CMMI® is rapidly establishing itself worldwide. The CMMI® continuous representation is gaining ground over the staged representation.

From a testing point of view CMMI® has more focus on verification and validation than CMM®, but testing maturity is still not addressed explicitly.

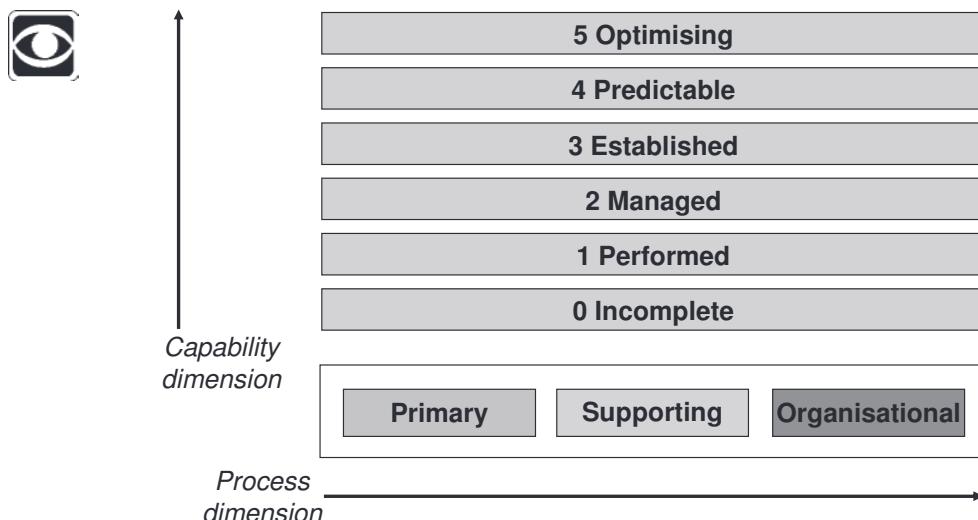
### 8.2.2.3 ISO 15504

The ISO 15504 standard for maturity assessments has long been known under the working name of SPICE, but it was finally released in 2003. ISO 15504 is mostly used in Europe and the Far East.

ISO 15504 is a continuous model (i.e., the capability is assessed for each of the defined process areas individually in an identical way). The result of an assessment is a capability profile.

The reference model defined in SPICE is two-dimensional. One dimension is the capability dimension with six capability levels. The other dimension is the process dimension structured in three process categories.

Each of the process categories is refined into a number of subcategories (called processes). The full structure is shown here.



#### PRIMARY

CUS Customer Supplier

CUS.1 Acquisition

CUS.2 Supply

CUS.3 Requirements Elicitation

CUS.4 Operation

**ENG Engineering**

- ENG.1** Development
- ENG.1.1** System requirements analysis
- ENG.1.2** Software requirements analysis
- ENG.1.3** Software design
- ENG.1.4** Software construction
- ENG.1.5** Software integration
- ENG.1.6** Software testing
- ENG.1.7** System integration and testing
- ENG.2** System and software maintenance

**SUPPORT**

- SUP Support**
- SUP.1** Documentation
- SUP.2** CM
- SUP.3** Quality assurance
- SUP.4** Verification
- SUP.5** Validation
- SUP.6** Joint review
- SUP.7** Audit
- SUP.8** Problem resolution

**ORGANISATIONAL****MAN Management**

- MAN.1** Management
- MAN.2** Project management
- MAN.3** Quality Management
- MAN.4** Risk Management

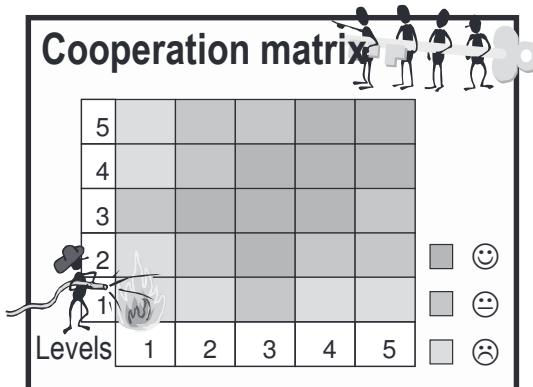
**ORG Organisation**

- ORG.1** Organisational alignment
- ORG.2** Improvement process
- ORG.3** Human resource management
- ORG.4** Infrastructure
- ORG.5** Measurement
- ORG.6** Reuse

ISO 15504 has much more focus on the development life cycle, and hence on testing, than do CMM® and CMMI®. This can be seen in the process category engineering where the subcategory development is further broken down into detailed development activities, including the processes software testing and software integration and testing.

### 8.2.3 Testing Improvement Models

This section begins with a word of warning. An old saying goes: "One should stick to one's own class." This is to some extent the case for organizations with regard to maturity. Research has shown that the maturity of organizations influences their ability to work together. The results are shown in the matrix here.



In general organizations at maturity level 1 are difficult to work with for organizations at level 1 and 2, and at level 4 and 5—even though the reasons are different. A level 3 organization can work with a level 1 organization, but it is not the best constellation.

The higher the maturity on both sides the better the cooperation, and more importantly, the more equal the maturity the better the cooperation.

This is worth keeping in mind for test organizations wanting to improve. The next section is about test-specific improvement models. Test organizations can mature using these models, but *the best result for the company as a whole is, if the different departments synchronize their process improvements.*



#### 8.2.3.1 TMM (Testing Maturity Model)

Dr. Ilene Burnstein, Institute of Technology, has said: "Testing is a critical component of the software development process. Organizations have not fully realized their potential for supporting the development of high-quality software products. To address this issue we are building a Testing Maturity Model (TMM) to serve as a guide to organizations focusing on test process assessment and improvement."

The Testing Maturity Model and TMM are service marks of the Illinois Institute of Technology

The following is based on two articles by Ilene Burnstein, Taratip Suwan-Nasart, and C.R. Carlson, Illinois Institute of Technology, published in U.S. Air Force magazine *Crosstalk*, August and September 1996 describing the TMM.

The Software Engineering Institute's Capability Maturity Model® (CMM®) does not adequately address testing issues. The TMM is built to overcome this.



The model is structured like the CMM®, but specifically addresses issues important to test managers, test specialists, and software quality assurance staff. Like CMM® the TMM contains a set of maturity levels, a set of recommended practices at each level of maturity, and an assessment model that will allow organizations to evaluate and improve their testing process.

The TMM can be used by a number of stakeholders:

- ▶ Internal assessment team to identify the current testing capability state
- ▶ Upper management to initiate a testing improvement program
- ▶ Development teams to improve testing capability
- ▶ Users and clients to define their roles in the testing process

The founders of the TMM point out that not only is the TMM structurally similar to the CMM®, it must be viewed and utilized as a complement to the CMM®, since mature testing processes depend on general process maturity.



In the development of the TMM a historical model provided in a key paper by Gelperin and Hetzel has been used. Their model describes phases and test goals for the periods of the 1950s through the 1990s.

Beizer's evolutionary model of the individual tester's thinking process in many ways parallels the Gelperin-Hetzell model. Its influence on TMM development is based on the premise that a mature testing organization is built on the skills, abilities, and attitudes of individuals that work within it. Beizer's phases in a tester's mental life are shown here.

Phase 4	= a mental discipline that results in low-risk software without much testing effort
Phase 3	= reduce the perceived risk of not working to an acceptable value
Phase 2	= show that the software doesn't work
Phase 1	= show that the software works
Phase 0	= support and debugging

The initial period in the Gelperin and Hetzel model is described as "debugging-oriented." During that period most software development organizations had not clearly differentiated between testing and debugging. Testing was viewed as an activity to help remove bugs.



In the "demonstration-oriented" period, a primary testing goal was to demonstrate that the software satisfied its specifications. Testing and debugging were still linked in efforts to detect, locate, and correct faults.

The "destruction-oriented" period focused on testing as an activity to detect implementation faults. Debugging was a set of separate activities needed to locate and correct faults.

In the "evaluation-oriented" period, testing became an activity that was integrated into the software life cycle. The value of review activities was recognized. The view of testing was broadened and its goals were to detect requirements, design, and implementation faults.

The Gelperin-Hetzell historical model is culminated by what they call a "prevention-oriented" period, which reflects the optimizing level 5 of both the CMM® and the TMM. The scope of testing is broadly defined and includes review activities. A primary testing goal is to prevent requirements, design, and implementation faults. Review activities now support test planning, test design, and product evaluation.



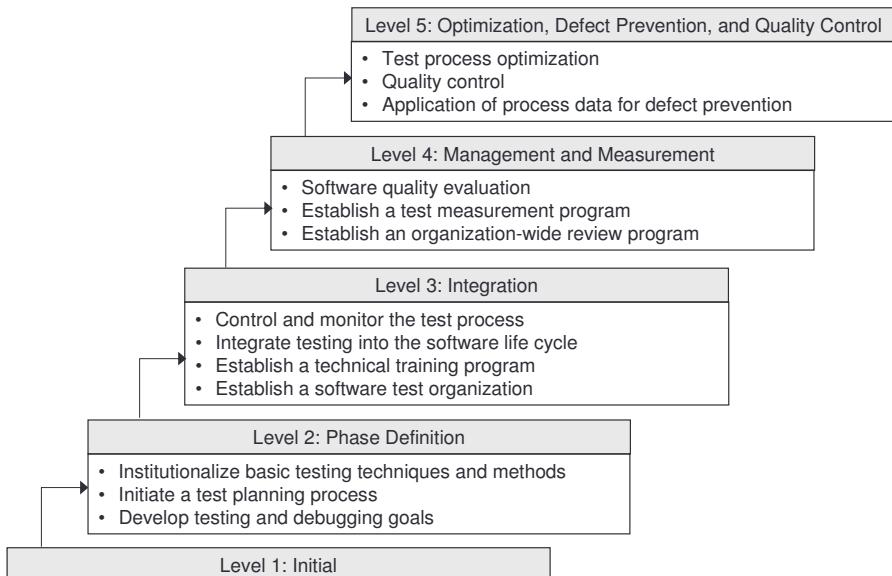
The attributes of a mature testing process, according to TMM, are:

- ▶ A set of defined testing policies
- ▶ A test planning process
- ▶ A test life cycle
- ▶ A test group
- ▶ A test process improvement group
- ▶ A set of test-related metrics
- ▶ Tools and equipment
- ▶ Controlling and tracking
- ▶ Product quality control

The TMM has two major components, a set of maturity levels and an assessment model.

### **TMM Maturity Levels**

Each of the levels prescribes a position in the testing maturity hierarchy. The characteristics of each level are described in terms of testing capability and organizational goals. They identify the areas where an organization must focus to improve its testing process. The hierarchy of testing maturity levels and goals is shown here.

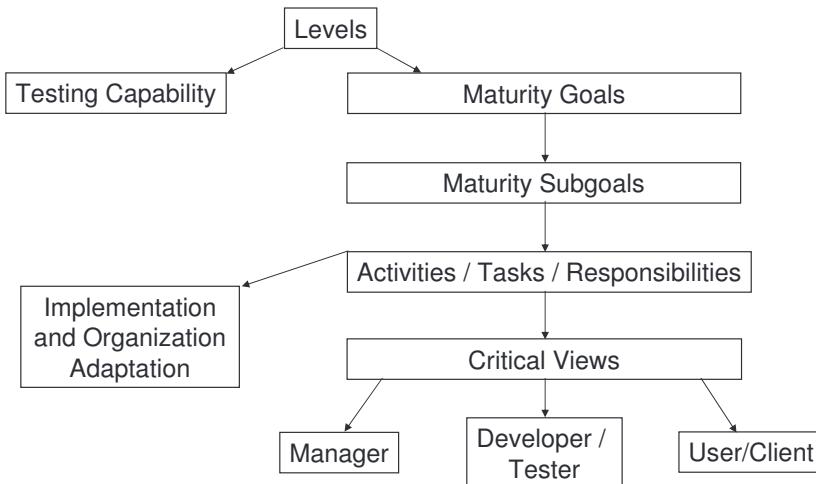




*Level 1 is the initial level. There are NO maturity goals at this level.*

Testing in level 1 organizations is typically a chaotic process, where tests are developed in an ad hoc way after coding is done. Testing and “debugging” are mixed to get the bugs out of the software. If there is a test objective, it is to show that the software works.

Each level above level 1 has a structure, much like the structure in CMM®:



- ▶ A set of maturity goals. The maturity goals identify testing improvement goals that must be addressed to achieve maturity at that level. Testing capabilities for each level reflect the goals.
- ▶ Supporting subgoals. They define the scope, boundaries, and needed accomplishments for a particular level.
- ▶ Activities, tasks, and responsibilities. They are necessary to achieve the goals associated with each level, and they have an influence on the implementation and adaptation.
- ▶ Responsibilities are assigned for these activities and tasks to three groups that we believe represent the key participants in the testing process: managers, developers and testers, and users and clients. In the model they are referred to as “the three critical views.”

The component called the “critical views” is added to the TMM in order for the testing process’s key participants to be included in process maturity growth.

The TMM is staged like the CMM® and requires that all of the capabilities at each lower level be included in succeeding levels.

**Ex.**

Level 2 is called “phase definition” and the maturity goals at this level are:

- ▶ Develop testing and debugging goals
- ▶ Initiate a test planning process
- ▶ Institutionalize basic testing techniques and methods

According to the structure each of the goals has a number of subgoals. It will be too much to go through the entire TMM model here, but we will take the second goal at level 2 as an example.

The complete set of *maturity subgoals for the level 2 goal “initiate a test planning process”* is:

- ▶ An organization-wide test planning committee must be established with funding.
- ▶ A framework for organization-wide test planning policies must be established and supported by upper management.
- ▶ A test plan template must be developed, recorded, and distributed to project managers and developers.
- ▶ Test work products must be defined, recorded, and documented.
- ▶ Project managers must be trained in the test planning process.
- ▶ A mechanism must be put in place to integrate user-generated requirements as inputs into the test plan.
- ▶ Basic planning tools must be evaluated and recommended, and usage must be supported by management.

### **TMM Assessment Model**

The assessment model in the TMM is composed of the following items:

- ▶ *The questionnaire:* This will contain questions that are designed to determine a level of testing maturity.
- ▶ *The assessment procedure:* This will give the assessment team guidelines on whom to interview and how to collect, organize, analyze, and interpret the data collected from questionnaires and personal interviews.
- ▶ *The Team Selection and Training Procedure:* The assessment procedure will be carried out by a trained assessment team internal to the organization being assessed.

### **TMM and CMM®**

TMM is developed as a complement to the CMM®. Organizations interested in assessing and improving their testing capabilities are likely to be involved in general software process improvement. To have directly corresponding levels in both maturity models would logically simplify these two parallel process improvement drives.

Research shows that an organization striving to reach the “*i*th” level of the TMM must be at least at the “*i*th” level of the CMM®. In many cases, a given TMM level needs specific support from key process areas in its corresponding CMM® level and the CMM® level beneath it. These key process areas should be addressed either prior to or in parallel with the TMM maturity goals.

A TMMI model aligned with the CMM® model is under development.



### 8.2.3.2 TPI (Test Process Improvement Model)

The TPI model is a dedicated test maturity model. It was developed in Holland in 1997 by Tim Koomen and Martin Pol.

The model was first developed because the authors needed a model to support their test-focused process improvement activities and were unable to find an existing model that satisfied their needs.

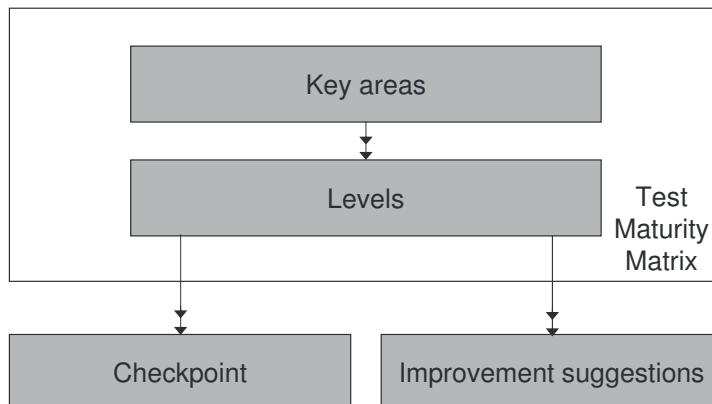
TPI is built on extensive testing experience and suggests the following improvement: After initial awareness of the general improvement ideas, the actual improvement work starts with an assessment. The result of this is used to define improvement actions. After planning and implementation of these actions, a new assessment can be performed to define the next actions. Test process improvement is an ongoing, never ending process.

The authors of TPI point out that test process improvement is only one of a much larger group of aspects that influence the total result of system development. Test process improvement should be aligned with other initiatives, such as general software process improvement and the total quality model (TQM).



#### *TPI Structure*

Like the other models, TPI has a well-defined structure. This is shown in the following figure.



The model defines a number of key areas and for each of these it defines a number of levels. For each level for each specific key area a number of checkpoints are defined. These checkpoints are used to determine if a specific level is achieved for the specific key area. To assist the test process improvement, a number of improvement suggestions are also defined for each specific level for each key area.

There are 20 key areas in all in the TPI, and these cover the total test process. The areas are:



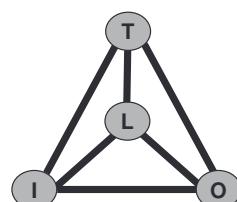
- ▶ Test strategy
- ▶ Life-cycle model
- ▶ Moment of involvement
- ▶ Estimating and planning
- ▶ Test specification techniques
- ▶ Static test techniques
- ▶ Metrics
- ▶ Test tools
- ▶ Test environment
- ▶ Office environment
- ▶ Commitment and motivation
- ▶ Test functions and training
- ▶ Scope of methodology
- ▶ Communication
- ▶ Reporting
- ▶ Defect management
- ▶ Testware management
- ▶ Test process management
- ▶ Evaluating
- ▶ Low-level testing

Martin Pol, one of the authors of TPI, has previously defined four cornerstones for the activities in a test organization in his TMap concept.

The cornerstones are:

- ▶ Life cycle
- ▶ Techniques
- ▶ Infrastructure and tools
- ▶ Organization

Each of the TPI key areas is assigned to a cornerstone.



### TPI Levels

As mentioned earlier, each key area has a number of levels. The levels are called A, B, C, and D. The levels are ordered so that it is always true that A < B < C < D in terms of time, money, and quality. The levels are individually named per key area.

The checkpoints are cumulative: If checkpoints for level A are met, then level A is achieved; to achieve level B all the checkpoints for both level A and level B must be met; and so on up to level D.

It should be noted that not all key areas have all levels.

The following table shows an overview of all the key areas and the levels defined for each. The levels are distributed on a scale ranging from 0 to 13. The relative position of the levels for the key areas is an expression of the dependencies between the key areas defined in the model.

The table also shows that the levels are grouped into three overall levels—controlled, efficient, and optimizing—vaguely corresponding to the CMMI® levels.



Key area	Scale												
	Controlled					Efficient					Optimizing		
	0	1	2	3	4	5	6	7	8	9	10	11	12
Test strategy	A					B				C		D	
Life-cycle model	A			B									
Moment of involvement		A				B			C		D		
Estimating and planning			A						B				
Test specification techniques	A	B				A	B						
Static test techniques				A		B							
Metrics					A			B			C		D
Test tools					A		B			C			
Test environment				A			B						C
Office environment					A								
Commitment and motivation	A				B					C			
Test functions and training			A			B			C				
Scope of methodology				A					B				C
Communication		A		B							C		
Reporting	A		B		C						D		
Defect management	A				B	C							
Testware management		A			B			C			D		
Test process management			A		B				C				
Evaluating						A		B			C		
Low-level testing					A	B	C						

**Ex.**

Let us take an example for one key area: test strategy.

- ▶ Cornerstone: L (life cycle)
- ▶ Description: The test strategy has to be focused on detecting the most important defects as early and as cheaply as possible. The test strategy defines which requirements and (quality) risks are covered by what tests. The better each test level defines its own strategy and the more the different test level strategies are adjusted to each other, the higher the quality of the overall test strategy.
- ▶ Levels:
 

<b>A:</b> Strategy for single high-level test	<b>B:</b> Combined strategy for high-level tests	<b>C:</b> Combined strategy for high-level tests plus low-level tests or evaluation	<b>D:</b> Combined strategy for all test and evaluation levels
---	--	---	--

All other key areas have similar specifications. Appendix 8.A provides a full list of the specifications for all the levels for all the key areas.

### **TPI Assessment**

As mentioned earlier, each level for each key area has a number of checkpoints to be met. The number of checkpoints varies from key specific level to key specific level.

There is no graduation applied in the model. This means that all checkpoints have to be fully met before a key area reaches a specific level.

In a TPI assessment the fulfillment of the checkpoints for level A is evaluated for each key area. For the areas where all the level A checkpoints are met, the fulfillment of the level B checkpoints is then evaluated – and so on until no more levels are achieved.

The checkpoints may be quite difficult to understand and a TPI assessment should be carried out by an educated assessor.

**Ex.**

The checkpoints for level A for the key area *test strategy* are listed here.

Test strategy for single high-level test:

- ▶ A motivated consideration of the product risks takes places, for which knowledge of the system, its use, and its operational management is required.
- ▶ There is a difference in test depth, depending on the risk and, if present, the acceptance criteria: Not all subsystems are tested equally thoroughly and not every quality characteristic is tested (equally thoroughly).

- ▶ One or more test specification techniques are used, suited to the required depth of a test.
- ▶ For retests also, a (simple) strategy determination takes place, in which a motivated choice of variations between “test solutions only” and “full retest” is made.

If you wonder how to interpret the first checkpoint, it may be translated to something like:

- ▶ Do you perform a risk analysis?
- ▶ Do you have rationales for the risks?
- ▶ Do the people involved in the risk analysis know the system and the ways it is going to be used?

The result of a TPI assessment is a test maturity matrix, which indicates how each of the 20 key areas has scored. This is the maturity profile of the test organization. The maturity profile shows the strong areas and the weak areas and provides the first indication of where to set in with improvement actions.

An example of a test maturity matrix is shown below. The light blue coloring is the result profile from an assessment.

### ***TPI Process Improvement***

The whole idea of TPI is test process improvement. The test maturity matrix should be used as the leverage for this.

The figure below shows an extract of a test maturity matrix. The light color indicates the result of the assessment.

Key area	Scale												
	0	1	2	3	4	5	6	7	8	9	10	11	12
Test strategy		A				B				C		D	
Life-cycle model	A			B									
Moment of involvement		A			B				C		D		
Test specification techniques	A		B										
Metrics				A			B			C		D	
Test tools				A			B		C				
Reporting	A		B		C					D			
Testware management		A		B				C				D	
Low-level testing			A		B		C						

The darker coloring indicates the target for the improvement activities. The aim is get the profile more smooth over the individual key areas.

The checkpoints defined for the levels for the key areas serve as an aid for improvements, because these checkpoints must be fulfilled. Furthermore the model includes a number of improvement suggestions (hints and tips) for each level for each key area.

It can be seen from the test maturity matrix above that the key area test strategy has already reached level A. The next target is to reach level B.

**Ex.**

The improvement suggestions for *test strategy* level B are summarized here.

Combined strategy for high-level tests:

- ▶ Obtain insight into what the different tests do
- ▶ Indicate possible risks
- ▶ Try to find obvious “holes” or duplicate testing
- ▶ Appoint a test coordinator
- ▶ Consider establishing an acquisition inspection (entry criteria for test)

### 8.2.3.3 CTPs (Critical Testing Processes)

In the beginning of this book we established that testing is process, and that the testing process can be broken down into more and more detailed processes.

The quality of the description of a given process and the way the process is performed influence the quality of the work done. Following well-described and well-fitting processes in a contentious way provides better results than do other combinations, such as following processes in a sloppy way (or not at all) or following ill-fitting processes.

The American Rex Black has introduced the concept of critical testing processing as a guideline for test process improvement.



The basis assumption is that some of the testing processes we can specify and follow are more critical than others. The criteria for a process to be defined as critical are that the process is:

- ▶ Repeated frequently
- ▶ Highly cooperative
- ▶ Visible to peers and superiors
- ▶ Linked to process success

The CTP model is a highly flexible model, and the aspects in it should be

tailored to the specific context in which it is being used. This includes that the organization using the model may identify its own specific challenges, and its own way of describing a process and defining the importance of the processes and hence the order of their improvement.

The critical processes must fit into the development model in which the testing is a support activity. Each development process must be linked to one or more testing processes and vice versa.



### ***The 12 Critical Processes***

Rex Black has defined the 12 processes he considers to be most critical. These are:



1. Testing
2. Establishing context
3. Quality risk analysis
4. Test estimation
5. Test planning
6. Test team development
7. Test system development
8. Test release management
9. Test execution
10. Bug reporting
11. Result reporting
12. Change management

For each of these processes the model explains what the heading covers and why the specific process is important.

One example is the first of the critical processes: *the testing process*. This process encompasses the activities:



- ▶ Planning: Determining what testing to do in the context
- ▶ Preparing: Designing and building the tests and forming the test team(s)
- ▶ Performing: Getting the test object, making it executable and testable, and testing it
- ▶ Perfecting: Reporting findings and guiding the process

This process is important, because the result of performing it well will be reducing costs by finding important bugs, providing useful information about less important bugs, reducing risks by identifying what works and what doesn't, and giving management essential information.

Another example is the *test estimation process*. This process encompasses the activities:

- ▶ Identifying the testing tasks, resources, and dependencies through a work breakdown structure
- ▶ Drawing on test and project team wisdom
- ▶ Putting together a budget
- ▶ Selling the estimate to management

This process is important because the result of performing it well will be balancing the cost and time required for testing against project needs and risks, forecasting the tasks and duration of testing in an accurate and actionable manner, and demonstrating the return on the testing investment.

### ***Assessment of the Testing Process***

The CTP model includes a guide for assessing the existing test process and its subprocesses in an organization. The assessment process itself may be tailored to the specific context.

The following top five points should however be assessed for the processes:

- ▶ Effectiveness
- ▶ Efficiency
- ▶ Pervasion
- ▶ Information provision
- ▶ Improvement

The findings for these points can be substantiated by both quantitative and qualitative measurements.

Examples of relevant metrics are shown here.

**Ex.**

	<b>Quantitative</b>	<b>Qualitative</b>
<b>Effectiveness</b>	Defect detection percentage	When and by whom are the worst bugs found
<b>Efficiency</b>	Return on the testing investment	Does testing seem worthwhile
<b>Pervasion</b>	Coverage of for example requirements and risks	Time of test involvement in the development lifecycle
<b>Information provision</b>	Number of reports provided to stakeholders	Test report utility
<b>Improvement</b>	Results of previous improvement activities	Perceived better quality and satisfaction



The measurements must be tailored to reflect the performance of each of the critical testing processes being assessed.

### ***Improving Critical Processes***

The result of a CTP assessment is a profile showing which processes are strong and which are weak. Taking the organizational needs into consideration this provides a prioritized recommendation of which process to improve at this particular point in time. There is no inherited recommendation for the order in which testing processes should be improved in the CTP model.

The improvement of the testing processes according to the CTP model is more or less identical to any process improvement task. A description of process improvement principles is found in Section 8.2.1.



#### **8.2.3.4 STEP (Systematic Test and Evaluation Process)**

The last model we are going to look at is in fact the first. The testing methodology called the STEP (Systematic Test and Evaluation Process) was developed by Drs. David Gelperin and William Hetzel in 1986. STEP was developed based on American National Standards Institute (ANSI) 829: Test Documentation Standard and 1008: Unit Test Standard. IEEE has since taken over these standards; this is why we now refer to IEEE 829, but it is the same standard.

At the time the STEP presented an entirely new concept, namely a test process to run parallel with the development and support validation of the completeness of each phase of development. The STEP model defines four test levels: acceptance test, system test, integration test, and unit test. It is a concept that holds to this day, as can seen from this book and virtually all other testing literature.

Before the STEP, testing was seen as something performed after the system was fully assembled. The STEP expanded that original definition of testing to encompass three main steps:

- 1 Plan the test strategy (develop a master test plan and associated detailed test plans)
2. Acquire testware (define test objectives, design and create test plans)
3. Measure (execute the tests, ensure that tests are adequate, and monitor the process itself)



The test planning activities are based on the ANSI Standard 829 requiring a master test plan and a detailed test plan for each of the test levels outlined in the master test plan. The detailed test plans are to be produced during the corresponding development phase.

In the test acquisition activity the STEP describes how test cases are to be produced to validate each requirement in a requirements-based testing approach for the higher test levels. At the lower test levels test cases are

produced to validate the design. The test cases are to be produced parallel with the development activities and serve as extra quality assurance of what the STEP refers to as inventory items (work products). Traceability matrices between test cases and inventory items must be produced.

The acquisition activity includes a risk analysis for prioritization of the test cases and the creation of test procedures, test data and tools, if any.

During test execution and measurement a test log is required, and incident reports must be made when incidents are discovered. At the conclusion of the testing a test report must be made; the findings and experiences made during the test must be analyzed and reported. The STEP outlines that the results of these activities, the documents, test data, and tools used are to be preserved for future use.

The STEP also covers maintenance testing including regression test.

In short we can say that the STEP supports the modern understanding of a good test process in that it promotes and supports:



- ▶ That testing starts at the beginning of the development life cycle
- ▶ That testers and developers work together
- ▶ A requirements-based test approach that testers use
- ▶ That tests are used as requirements and usage models
- ▶ That testware design leads software design
- ▶ That defects are detected earlier or prevented all together
- ▶ That defects are systematically analyzed

The STEP is not a process improvement model in the sense discussed above. It presents a test process, whose introduction in an organization is the improvement. The way the STEP process is used may be improved by auditing (assessing) the process and analyzing the measures collected. These measures include:

- ▶ Test status over time
- ▶ Test requirements or risk coverage
- ▶ Defect trends
- ▶ Defect density
- ▶ Defect removal effectiveness
- ▶ Defect detection percentage (test effectiveness)
- ▶ Defect introduction, detection, and removal phase(s)
- ▶ Cost of testing

More quantitative factors may include:

- ▶ Defined test process utilization
- ▶ Customer satisfaction

## Questions

1. What is a “standard”?
2. What may the sources of standards for use be?
3. Which are the two most widely known standard “families”?
4. Which standard covers product quality aspects?
5. Which standard covers software review and audit?
6. What is the disadvantage of domain-specific standards?
7. Why is there a growing need for software process improvement?
8. Where can the pressure for process improvement come from and why?
9. What are the basic principles in process improvement?
10. Which role do tools have in process improvement?
11. Which companies are most likely to succeed with process improvement?
12. What characterizes more mature organizations?
13. What does CMM® mean?
14. What are the names of the five maturity levels in CMM®?
15. What is a key process area?
16. Which two representations are defined for CMMI®?
17. What are the four groups of process areas in CMMI®?
18. How is testing represented in CMMI®?
19. What is the ISO number of the SPICE model?
20. What are the three process categories defined in SPICE?
21. What does TMM mean?
22. What are the phases in a tester’s mental life according to Beizer’s?
23. What are the five levels in TMM?
24. How is the structure for a level in TMM?
25. What does the TMM assessment model consist of?
26. How do TMM and CMM® relate to each other?
27. What does TPI mean?
28. What is the structure of the TPI model?
29. How many key areas does TPI define?
30. What are they?
31. What are the groups into which the levels are organized?
32. What is done in a TPI assessment?
33. What is the result of a TPI assessment?
34. How is the result used for process improvement?
35. What does CTP mean?
36. What makes a process critical?
37. What critical processes does the model define?
38. What should be assessed in a CTP assessment?
39. What does STEP mean?
40. What is the STEP based on?
41. What are the three main steps in the STEP?
42. How is improvement addressed in the STEP?

## Appendix 8A Definition of Levels in the TPI Model

CS = Cornerstone; levels are not defined for blank key area/level cells.

<b>CS</b>	<b>Key Area</b>	<b>Level A</b>	<b>Level B</b>	<b>Level C</b>	<b>Level D</b>
I	Test tools	Planning and control tools	Execution and analysis tools	Extensive automation of the test process	
I	Test environment	Managed and controlled test environment	Testing in the most suitable environment	"Environment-on-call"	
I	Office environment	Adequate and timely office environment			

L	Test strategy	Strategy for single high-level test	Combined strategy for high-level tests	Combined strategy for high-level tests plus low-level tests or evaluation	Combined strategy for all test and evaluation levels
L	Life-cycle model	Planning, specification, execution	Planning, preparation, specification, execution, and completion		
L	Moment of involvement	Completion of test basis	Start of test basis	Start of requirements definition	Project initiation
T	Estimating and planning	Substantiated estimation and planning	Statistically substantiated estimation and planning		
T	Test specification techniques	Informal techniques	Formal techniques		
T	Static test techniques	Inspection of test basis	Checklists		
T	Metrics	Project metrics (product)	Project metrics (process)	System metrics	Organization metrics (> 1 system)

CS	Key Area	Level A	Level B	Level C	Level D
O	Commitment and motivation	Assignment of budget and time	Testing integrated in project organization	Test-engineering	
O	Test functions and training	Test manager and testers	(Formal) methodical, technical and functional support, management	Formal internal quality assurance	
O	Scope of methodology	Project specific	Organization-specific	Organization optimizing, R&D activities	
O	Communication	Internal communication	Project communication (defects, change control)	Communication within organization about the quality of the test processes	
O	Reporting	Defects	Progress (tests, products, costs, time, milestones, defects with priorities)	Risks and recommendations, substantiated with metrics	Recommendations have a software process improvement character
O	Defect management	Internal defect management	Extensive defect management with flexible reporting facilities	Project defect management	
O	Testware management	Internal testware management	External management of test basis and object	Reusable testware	Traceability system requirements to test cases
O	Test process management	Planning and execution	Planning, execution, monitoring, and adjusting	Monitoring and adjusting within organization	
O	Evaluation	Evaluation techniques	Evaluation strategy		
O	Low-level testing	Low-level test life-cycle (planning, specification, and execution)	White-box techniques	Low-level test strategy	



## CHAPTER

# 9

## Testing Tools and Automation

The purpose of using tools for testing is to get as many as possible of the noncreative, repetitive, and boring parts of the test activities automated. The purpose is also to exploit the possibility of tools for storing and arranging large amounts of data.

There are a huge number of testing tools on the market, and it is growing fast. Have a look in Appendix 9A! Every testing tool automates some testing activities to a certain degree. No single tool automates everything completely. But there are testing tools for all testing activities, even though most testers think about test execution tools when test automation is mentioned.

Test automation is not an easy task. A company can be more or less ready for test automation. It requires a certain level of maturity to be able to use tools efficiently. Tools do not provide more maturity; they should be implemented to support the existing maturity.

It also requires a certain amount of courage to engage in test automation, both courage to choose and to refuse. It is important to select tools with great care so that they don't end up as "shelfware."

It may be difficult to choose, but that is peanuts compared to getting a tool introduced in an organization. And keeping it running efficiently is perhaps even more difficult.

### Contents

9.1 Testing Tool Acquisition

9.2 Testing Tool Implementation and Deployment

9.3 Testing Tool Categories

## 9.1 Testing Tool Acquisition

Many tools are bought in excitement. We find a tool—on an exhibition or in a magazine – and we are immediately convinced that this tool is the solution to all our problems. The tool is used for a short while; it appears that it was not quite what we expected; and sooner or later it ends up on a shelf and is completely forgotten about. Sound familiar?

In a professional organization it is important to treat the investment in (testing) tools as the serious decision it is. Tools are usually expensive, and even if they are not expensive to buy, they are expensive to implement and maintain in the organization.

Acquisition and introduction of a tool in a company requires organizational considerations. It is not something you just rush in and do (like fools!); conscious decisions about what to do and how to do it in the company must be made before the work can commence.

The acquisition should include the following activities:

- ▶ Tool selection preparation
- ▶ Tool evaluation
- ▶ Selection of the winner

### 9.1.1 Tool or No Tool?

The first thing we must do when the idea of automation occurs is find out what it is we are trying to achieve with the tool. What exactly is the problem?

Introduction of a testing tool or testing automation is not necessarily the answer to all problems.

If the problem is that we are not entirely sure how to perform a task or an activity, it may be tempting to get a tool to help us, but it is usually not a good idea. Only work that is well-specified is appropriate for automation.

Work that requires creativity is not a candidate for automation either. We cannot get a computer to be creative and think outside of the box.

Automation may help solve problems caused by:

- ▶ Work that is to be repeated many times
- ▶ Work that it is slower to do manually
- ▶ Work that it is safer to do with a tool

Once the problem is described and wellunderstood, we can consider how to solve it. There may be a number of alternative solutions, including the acquisition of a tool.

Maybe it does seem like a tool is the best solution, and in that case we can go on with the selection preparation.

A fool with a tool  
is still a fool

### 9.1.2 Tool Selection Team

The next step is to establish a team to perform the evaluation and selection of the tool. This team must be as broad as possible and include representatives for all potential stakeholders for the test tool.

The team must be composed of a team leader and representatives for all potential users of the test tool, including developers, professional testers, responsible for tools, responsible for process, and future product users.

The team members cannot be expected to be assigned full-time to the selection and evaluation task. It must, however, be ensured that they are available when meetings are held and take an active part in the work.

### 9.1.3 Testing Tool Strategy

The long-term testing tool strategy in the organization must be considered if it is already in place, or it must be produced if it is not.



The point to make clear is how a new tool will fit into the overall goals for the company (e.g., with regard to general process improvement or the achievement of a certain level of capability or a specific certification). This may have an impact on the type of tool to choose.

It must also be clear how large a part of the organization is going to use the new tool. Will it be all so that we should choose a solution that covers the entire company; or will it be on a project level so that we should choose a solution that only covers the needs in a single, independent project? This decision may have far-reaching consequences both with regard to direct cost and with regard to time to be spent.

At some point it is of course also important to establish who is going to pay for a new tool and for the continued usage and maintenance.

### 9.1.4 Preparation of a Business Case

In a business case we compare the cost of a solution with the benefits the solution is going to bring us.

The cost of selecting, implementing, and maintaining a tool is usually significant. It includes expenses for:

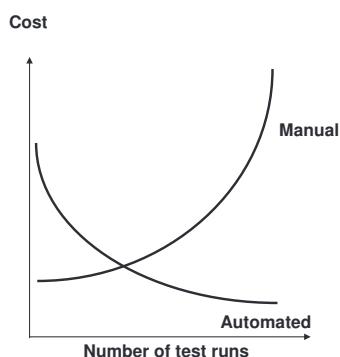


- ▶ Selection
- ▶ Acquisition (list price minus possible discounts, open source, or own development)
- ▶ Licenses
- ▶ Tailoring
- ▶ Implementation
- ▶ Training
- ▶ Tool usage
- ▶ Maintenance of automated testware
- ▶ Tool maintenance

Some of these expenses are measured directly in money; others come from time being spent by employees; both must be considered in the calculation.

On the other side of the business case equation, we have the benefits. Benefits from test automation are rarely, if ever, measurable in actual money. They come from savings we obtain because the tool helps us perform the tasks faster and with fewer mistakes.

For test execution tools the cost/benefit depends very heavily on how often the automated tests will be executed, as illustrated here.



Tests that are only executed a few times during the entire lifetime of the product are usually not worth spending automation resources on.

On the other hand, it may be well worth automating the tests that are executed many times, for example, tests used for extensive regression testing of high-risk areas.

It is of course possible to have a mixture of manual and automated tests.

### 9.1.5 Identification of Tool Requirements

A testing tool is a software product, and just like all other software products it should satisfy a number of requirements. It is part of the selection preparation to identify the requirement applicable for the tool to be implemented.

First of all the tool needs to have some *functionality*. We must define what we require the tool to do. This must of course be consistent with the processes we are going to automate. The integration with other tools is another important part of the functionality we need to specify. If the tool is going to have Web access, this must also be explicitly and thoroughly described.

Connected to the functional requirements we have the *nonfunctional requirements*. These should at least include aspects of performance, usability, availability, and maintainability for the tool.

There are many aspects of performance, and these aspects may have a greater impact on everyday life than you may think. It may, for example, be the time it takes to execute a complete regression test suite, or the volume of something that the tool can handle.

Usability is a measure for how easy a test tool is to use. It may include aspects like intuitive interface, help facilities and user documentation, compatibility with existing procedures, and tailoring facilities.

Availability describes when we can use the tool. If the tool is running on a server somewhere this can be fairly unpredictable.

We should also consider availability of support. Is it possible to get support in our own language during normal working hours, or do we have to call somewhere in the middle of night and try to explain our problems in a foreign language?

Maintainability is interesting, for example, in terms of upgrades. How often do we get new versions? Will they be backwards-compatible? What about our own tailoring?

The *environmental requirements* or constraints are requirements forced on us from the environment around our organization. It can be in the form of existing products that the testing tool must be able to cooperate with or a specific platform that we need to use.

The last thing to consider is *project requirements* or constraints. They are the usual ones, namely resources, time, and money. They form the foundation of the requirements tower and they need to be “strong” enough to carry the tower. If not we must reduce the product quality requirements or increase the project requirements.

### 9.1.6 Buy, Open-Source, or Do-It-Yourself

There are advantages as well as disadvantages, to buying a standard tool, getting an open-source tool, or developing one's own tool. This is a consideration worth making, and it may be based on the aspects shown here.

Buy	Open-Source	Do-it-Yourself
Some tailoring must always be foreseen, either to the tool or to the processes in the company, or both.	The tool may be changed and enhancements should be shared.	The tool can be made exactly as the company wants it (provided it knows what it wants).
The price is usually easy to calculate.	The tool is free but there may be license fees to pay.	It may be even extremely difficult to estimate the final cost.
Usually the payment must be made within a relatively short period of time.	No immediate price needs to be paid.	The development and hence the “payment” can be done at the company's own pace.
Do what you do best – that is what the suppliers do.	The quality depends on the exposure, history, and use of the tool.	Maybe you are the best suited to develop your own tool.

The lists are by no means exhaustive, but it may be used for inspiration for the considerations.

If a company decides to develop its own tool, this must be undertaken like any other development project (i.e., at least as seriously as a project with an external customer). This will not be discussed further.



We need to be aware that many tools come into existence because developers or testers have an urgent need for tool support for a specific task and are able to develop a tool themselves. This is often done as “hidden” work, (i.e., the time spent is not registered anywhere). Such tools may be very efficient and they may be taken into consideration when selecting a more official tool solution. It must be evaluated if such tools are sufficiently documented to build on, and if they can handle the scaling involved in spreading the use to a larger user group.

### 9.1.7 Preparation of a Shortlist of Candidates

Many sources for information about testing tools exist, for example, articles, suppliers’ Web pages, other companies, exhibitions, and research reports.

Based on the first information a number of possible candidates are identified by a fairly coarse evaluation method based on some really essential requirements like the platform on which a given tool may run.

It can be useful to supplement the evaluation with a look at the supplier. The supplier is “the family-in-law” that we will have to live with for a long time, so investigate for example:



- ▶ The supplier’s employees—Do they match ours?
- ▶ The supplier’s own use of the tool
- ▶ The supplier’s financial status
- ▶ The supplier’s focus—Is testing tools a niche?
- ▶ The supplier’s acquaintances
- ▶ The supplier’s reputation
- ▶ The supplier’s support facilities

### 9.1.8 Detailed Evaluation

After this first selection a stricter and stricter evaluation is made until only two candidates are left in the field.



It is important that the evaluation group agrees on how the evaluation is to be made and precisely what is significant in the selection. An evaluation method includes:

- ▶ Description of the scale for the evaluation of fulfillment of the requirements, for example
  - ▶ Fully, Almost, Partly, Not
  - ▶ From 0 to 100%

- ▶ Description of the selection criteria, based on the fulfillment evaluation, for example
- ▶ All priority 1 requirements fulfilled at least 80% and at least 50% of the priority 2 requirements fulfilled

It is a good idea to define different criteria for different selection phases, typically more strict as the field narrows.

After some evaluation rounds the list of candidates has been reduced from a number of possible test tools to fewer and fewer candidates by the deployment of the defined evaluation method. In the end there should be only two left.

### 9.1.9 Performance of Competitive Trials

The two finalists should undergo a detailed evaluation that should include at least one demonstration and preferably a trial period, so that the tools may be tried out under as realistic circumstances as possible.

Scenarios that reflect the functional requirements should be set up and run through. At the same time the nonfunctional requirements can be tested. Performance aspects should be evaluated under realistic circumstances, that is, both locally and over great distances, if that is the need, and in an environment with the “normal” load, not just on an isolated test machine.

It may be important to investigate if a tool can handle the volumes that the company or the project may have to handle. Volume may also be a question of a large number of users and/or a large number of platforms possibly distributed over large distances. It should not only be the company’s current situation that is included in an evaluation; a testing tool should be able to cope with the development in the company for at least the foreseeable future.

For testers it should not be difficult to test a tool they are going to use themselves. Even if we are really eager to start using testing tools it is worth remembering that advertising material and salespeople may color things a bit.

The last thing to do for now is to



## select the winner!

### 9.2 Testing Tool Introduction and Deployment

Now the tool has been selected. The real challenge, however, is to make the tool become part of everyday life, and to keep it alive long enough to profit on the investment.

The introduction or implementation of a tool in an organization is an organizational change project. The principles of process improvement in general are discussed in Section 8.2.1.



Management commitment is essential for the implementation to be a success. An implementation process must be described and followed closely to avoid the tool ending up as “shelfware,” as so many tools unfortunately do.

An implementation process should include the following activities:



- ▶ Make necessary adjustments
- ▶ Perform a pilot project
- ▶ Assess the pilot project
- ▶ Produce a rollout strategy
- ▶ Make the rollout happen
- ▶ Follow up on the rollout

The necessary resources, both in terms of people, time, money, and training must be provided and sustained until the usage of the new tool is an engraved part of everyday working life.

”

The activities needed in implementation of new ways of working are described in detail for the introduction of static testing in Section 6.4. The principles are the same for the introduction of a tool, and the activities are only summarized here.

The roles that must be in place to make the tool implementation a success are:

- ▶ The sponsor
- ▶ The target group
- ▶ The champions
- ▶ The change agents

Furthermore the introduction of a tool requires a *tool custodian*. This is a technical person who is responsible for the setup and maintenance of the tool. He or she provides internal help and support with technical issues and can be responsible for contact to the supplier of the tool for second-level support.

### 9.2.1 Testing Tool Piloting

A pilot project should always be performed for the tool before we commit to implementing it across all projects.

It may be that there are some adjustments or tailoring to do, before the pilot can start. One hopes that the tool that has been chosen complies with the existing processes, but there may be smaller discrepancies.

This tailoring can be anything from making adjustments to make the tool comply completely to the processes to using the tool as it is, that is, tailoring the processes to the tool. Once the tool is bought the easiest and most future-safe thing to do is the latter – tailoring the processes to the tool.

There are a number of reasons for performing a small-scale pilot project. First of all we need to verify the business case and ensure that the benefits of the usage of the testing tool can really be achieved.

A goal for the pilot project is also to get some experience in the usage of the testing tool. The pilot should enable us to identify further adjustment need to the processes and to the tool, as appropriate. The different tools of all the different tool types support different detailed processes. They also require interfaces with other tools and other processes, for example, configuration management of testware. Finally a pilot can help us refine the estimate for the actual costs and benefits for the implementation.

A pilot should take between three and six months and be followed closely.

### 9.2.2 Testing Tool Rollout

The rollout of the testing tool should be based on a successful evaluation of the pilot project. Rollout normally requires a great involvement of all the people carrying roles in the test tool implementation, not least the users of the testing tool, the target group.

A rollout strategy that suits the nature of the organization must be defined. A “big-bang” rollout, where everybody starts using the tool at a given point in time, works in some organizations. In other organizations a gradual implementation, where the tool is deployed as the need arises, will work better.

No matter how the rollout is done the most important activity at this point is to support the new users as the rollout takes place. We must be prepared to

- ▶ Support the users



until the usage of the testing tool is a completely integrated part of the work.

### 9.2.3 Testing Tool Deployment

A testing tool is a part of the test environment for our tests, and in many ways like any other (software) product. The tools we use should be kept under proper configuration management like the rest of the test environment and other testware.

It is important to be able to register with which version of a tool specific tests have been prepared and/or executed. There is more about the concepts of configuration management, especially for testers, in Section 1.1.3.



## 9.3 Testing Tool Categories

### 9.3.1 Testing Tool Classification

Many tools for the support of software development are available, and the selection is growing every day.

It is therefore impossible to list specific tools. The purpose of this section is to present different types of testing tools and give an idea of the advantages and possible disadvantages of them.

Tools may be classified to get a better overview of the tools available. There are different classification schemes, for example according to:



- ▶ The test activity they support
- ▶ The test level the tools primarily support
- ▶ The types of failures or defects they can find
- ▶ The test approach or test technique they support
- ▶ The purpose they have
- ▶ The domain to which they are applied
- ▶ Who the primary users of the tools are

*The last categorization is used here.*

Tool support exists for the following primary users:



- ▶ All testers
  - ▶ Test management tools, including configuration management tools
- ▶ Test analysts and technical test analysts
  - ▶ Test design tools
  - ▶ Test data generation tools
  - ▶ Test oracles
  - ▶ Simulation and emulation tools
  - ▶ Test execution tools
  - ▶ Keyword-driven automation tools
  - ▶ Comparison tools
  - ▶ Fault-seeding and fault-injection tools
  - ▶ Web tools
- ▶ Technical test analysts only
  - ▶ Static analysis tools
  - ▶ Dynamic analysis tools
  - ▶ Performance testing tools
- ▶ Programmers (or technical test analysts writing and maintaining test scripts)
  - ▶ Debugging, tracing, and troubleshooting tools

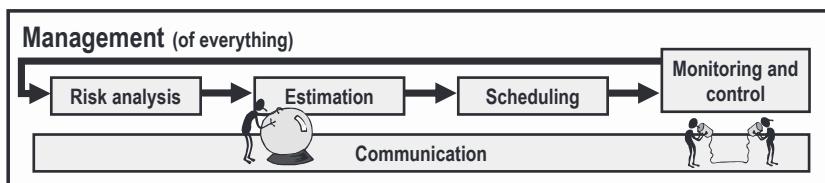
Not all the tools in the areas listed here are testing tools in a narrow sense, but they are all useful in testing and hence included in this overview. It must be stressed again here, that debugging is NOT a test activity, though tightly connected to testing, especially low-level tests.



### 9.3.2 Tools for All Testers

#### 9.3.2.1 Test Management Tools

Test management, like all management, includes risk analysis, estimation, scheduling, monitoring and control, and communication. Test management is discussed in Chapter 3.



Test management tools cover these activities and support the project management aspects of testing. These tools can typically be used for registration of test activities, estimation, scheduling of tests, logging of results, and analysis and reporting of progress.

Most test management tools provide extensive reporting and analysis facilities.

Test management tools can support the handling of test documentation, such as plans, test specifications, and test procedures, and even traces between test cases and requirements.

The *advantage* of test management tools is that they can assist in the management of all the activities in testing. They can provide an overview of the testing task and show progress.

There are *no direct disadvantages* of test management tools. They are, however, often wedged between other tools, such as project management tools and configuration management tools. Most test management tools provide some of the facilities that these other tools also provide. There are hence often many interfaces and/or redundancies in connection with other tools used in the organization. The reason for this is—to some extent at least—is that no single management tool provides all the needed features for software development management.

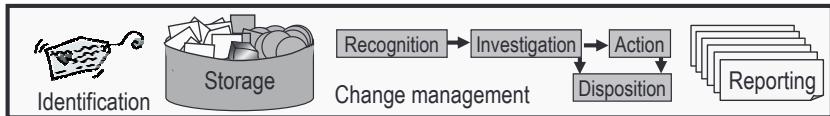
Another reason for the confusion between tools is that the borderlines between project management, configuration management, and test management are often blurred or not defined. This has got to do with the maturity of the organization. The more mature an organization is, the more the individual process areas are understood and clearly defined, and the easier it is to define what the tools should do and not do.



### **Tool Support for Configuration Management**

“

Configuration management is identification, storage, change management, and reporting of configuration items. Configuration management for testers is discussed in Section 1.1.3.



Configuration items are all work products, product components, and components that we want to control. This includes testware, such as test plans, test specifications, and test environments including tools and test results; and it includes requirements. Requirements are essential for testing, and it is therefore of special interest to testers how requirements are managed.

Configuration management tools are used to support the configuration management activities. The main features of these tools are:



- ▶ Identification and storage of items
- ▶ Traceability between items
- ▶ Incident reporting and management of the life cycle of faults
- ▶ Reporting and analysis

Traceability and incident management are important features. Requirements and test cases should be traced to each other, and traceability tools allow the link between test cases and their corresponding test coverage items to be recorded. Changes to requirements must be communicated to testers and appropriate consequential changes implemented, for example, in related test cases. This is facilitated by trace information.

Changes to configuration items should always be initiated by an incident report, and the main supplier of incident reports is testing. There is therefore a strong interface between testing and configuration management. Only a few configuration management tools include full change management, but a large number of more or less independent tools exist for this. Incident management tools (also known as defect tracking tools) may also have workflow-oriented facilities to track and control the allocation, correction, and retesting of incidents.

The main *advantage* of these tools is that they support the cumbersome and difficult information administration associated with the configuration management activities. Configuration management is difficult, if not impossible, to perform without some sort of tool support.

Most configuration management and incident management tools support analysis and reporting of configuration management information. This facilitates communication of the facts about how the development and testing processes are working.

The *pitfall* of these tools is redundancy in features and information and/or the need for transfer of data between tools, caused by unclear borders between tools, as described earlier.

### 9.3.3 Tools for Test Analysts and Technical Test Analysts

#### 9.3.3.1 Test Design Tools

Test design tools support the creation of test specifications. They can analyze a specification of the product, often expressed as a model in a formal way, and generate high-level test cases and possibly test procedures or scripts based on this analysis.

This type of testing tool can, for example:

- ▶ Derive high-level test cases from formally specified requirements, often managed by the same tool
- ▶ Generate test cases based on the specification of a model, for example, UML or state machines
- ▶ Generate input for test cases based on input models, for example, input distribution specifications
- ▶ Derive high-level test cases from actual source code

The *advantage* is that test cases are systematically and comprehensively derived from the basis documentation. If the basis documentation is produced in accordance with specified rules, no test case will be missed, and they will all be correct.

The *pitfall* is that these testing tools only do half (or less) of the work. They cannot specify the expected results, so we have to elaborate the test input provided by the tool into test cases with the definition of the expected result and preconditions.

The test design tools require very formally formatted basis documentation, and that can be regarded as both an advantage and a disadvantage.



#### 9.3.3.2 Test Data Preparation Tools

Test input data preparation tools support:

- ▶ Selection (e.g., from an existing database)
- ▶ Creation
- ▶ Generation
- ▶ Manipulation
- ▶ Editing



of test data for use in setting up preconditions for test procedures and individual test cases.

Some of these tools are data tool-dependent, while the most sophisticated can deal with a range of file and database formats.

Test data can be selected and extracted from live data and scrambled to hide person-sensitive information. This enables tests to be performed on real data, something that can be essential for systems in, for example, the public sector.

### Ex.

A test data preparation tool is able to extract live data from the tax authorities' database according to specific selection criteria for test runs of the implementation of a new tax law. The criteria may be 100 families with one income and at least three children, 100 people over 80 years of age with an income over a certain amount, and the 40 people with the highest income in a specific city. The tool scrambles the information that can identify the people in the test data (e.g., Social Security number, before the data may be used).

The *advantage* is that these tools make it possible to handle great volumes of data.

Usage of this type of testing requires good configuration management of testware to identify which specific versions of the object, the test specification, and the test data belong to each other.

The *pitfall* here is that the tools may create too much useless data, if selection is not planned carefully.

#### 9.3.3.3 Test Oracles



A test oracle is a special concept in test automation; it is used to determine expected results from inputs. Some say that the best test oracle is the tester, studying the test basis documentation and deriving the expected result from this. This is, however, sometimes not possible for time and/or cost reasons.

*Automated test oracles* are tools that can generate the expected result for specific input and hence facilitate the creation of test cases. Such "oracles" are hard to find. In principle they must do exactly the same as the object under testing and may therefore seem redundant.

One of the situations where an oracle can be found and can be very useful is when an old system is being replaced by a new one providing the same functionality. This is seen more and more often when old legacy systems are replaced with systems using new technology, for example, Web access. In such a case test input may be given to the old system and the result be regarded as the expected result for the new system.

Oracles can also be created in situations where nonfunctional requirements can be disregarded and a system simulating the functionality only can be developed at a much lower cost. This is especially the case where the real system has very strict performance requirements.

The *advantage* of oracles is that they make it possible to generate the expected results much faster than if we had to derive them manually. The use of oracles requires strict control over the oracle and the other testware.

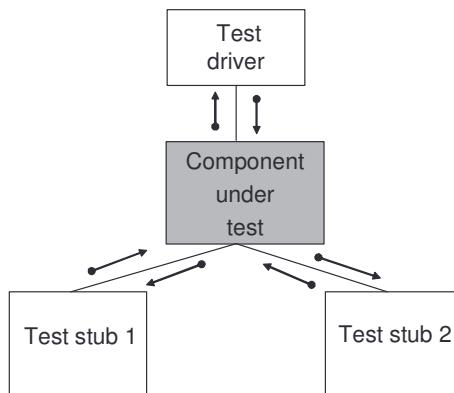
The *disadvantage* of oracles is that their usage can give us a false sense of reliability. There is a risk that we repeat faults in the old system, or between an oracle system and the real system. There is also a risk of not getting sufficient test coverage.

#### 9.3.3.4 Simulation Tools and Emulation Tools

Simulators are used to support tests where necessary code or other systems are either unavailable or impracticable or even dangerous to use. I for one would rather not test the software intended to handle a nuclear meltdown under real-life conditions.

Test harnesses and drivers fall into this category of tools. They are used where components or other test objects cannot be executed directly. It can be for testing of a component in isolation, embedded software without a user interface, or execution of many unrelated automated test scripts.

Some testing tools on the market provide harness and driver facilities, especially component testing tools. Very often, however, these tools are homemade and tailored precisely to needs. The principle in stubs and drivers is illustrated.



A special type of simulators is called emulators because they are used to mimic hardware to which the software under testing interfaces.

Simulation tools are almost always bespoke systems made for a specific assignment.

The *advantage* of these tools is that they make otherwise impossible or difficult tests possible. These tools can save us a lot of money.

Emulators can make it possible to test in “slow motion,” and they can act as debuggers as well.

These tools, like most other testing tools, require good configuration management of the testware, something that may be considered an advantage.

The *disadvantages* include that the use of these tools may give a false sense of reliability—after all the simulators or emulators may be wrong. It may also be that the usage of such tools “hide” defects, for example, performance and other time-related defects. Simulators and emulators can also be rather expensive to produce and set up, and the cost must be balanced carefully with the benefits.

Another disadvantage of these tools can be that they require that the testers can code or have access to people who are able to do the coding. In practice this is not a problem, since these tools are often used in testing being performed in close connection with the coding.

### 9.3.3.5 Test Execution Tools

This type of testing tool goes under many names: test execution tools, or test running tools, or capture and replay tools, and is probably the most widespread category of testing tool.



These tools are primarily used for automation of regression testing. They can execute test scripts much faster and more reliably than human beings, and they can therefore reduce test execution time when tests are repeated and/or allow more tests to be executed.

All the tools of this category work according to the same basic principles, namely:

- ▶ *Capture*: A recording of all the tester’s manual actions and the system’s responses into a test script
- ▶ *Control points*: A number of checkpoints added to the script by the tester during the capture
- ▶ *Playback*: Automatic (re)execution of the test script

Test execution tools exist for graphical user interface, GUI, and for character-based interfaces. For GUI applications the tools can simulate mouse movement and button clicks and can recognize GUI objects such as windows, fields, buttons, and other controls.

When a script has been captured once, it may be executed at any given time again. If the software under testing reacts differently from what was expected at the inserted checkpoints, the execution will report a failure. It is usually also possible to log information during execution.

Test scripts are captured in a specific scripting language. In newer versions of these tools it has become possible to get access to the scripts that have been captured. These are often in a C or Visual Basic like code, and this offers the possibility for editing the scripts, so that, for example, forgotten operations, further control points, or changed values, may be added.

Experience shows that if the scripts are written from scratch, rather than captured, and good and systematic development principles are used, the scripts will be more maintainable. More and more of these tools are therefore used as test execution tools of coded scripts, rather than capture/playback.

The *advantage* of these tools is that a lot of manual test execution can be done automatically. This is especially the case in iterative development and other development projects where a large number of regression testing are needed. These tools are indispensable in development where “frequent build and smoke test” principles are used. Builds can be made in the evening, and automated test suites can be set to run overnight. Testing results will then be ready in the morning.

The use of test scripts requires good configuration management to keep track of which versions of the test objects, test data, and test scripts belong together. Again a blessing (maybe) in disguise.

The *pitfall* for these tools is that it can be rather expensive to establish and maintain the test scripts. The requirements, specifications, and code undergo changes in the course of the development, especially in iterative development. This must be carefully considered in connection with the estimation of the continuous maintenance of the test scripts.

Another pitfall is that the work with the test scripts requires programming skills. If the necessary skills are not available, the use of test execution tools may be very cumbersome and inefficient.

At the same time it must be kept in mind that test scripts written by a programmer or tester are just like any other (software) product: made by humans and therefore not perfect. Defects are also introduced in test scripts, and test scripts should therefore be tested and corrected when defects are identified. The earlier this is done the better since fewer defects in the test scripts reduce the possible uncertainty as to whether a failure is caused by a defect in the test script or indeed in the product under test.



### 9.3.3.6 Keyword-Driven Automation Tools

Keyword-driven test is a way to execute test scripts at a higher level of abstraction. The idea is similar to that of a service or subroutine in programming where the same code may be executed with different values.

Keywords are defined to represent a script, and a tool can then act as a link between the keywords and the tool executing the corresponding test script. Values may be assigned for parameters associated with the keywords.

The tools make it possible to use parameter-driven test scripts without having to change the (often complicated) scripts in the execution tool.

Keywords are usually related to higher level functionality or business procedures. They may also reflect use cases.

The tools for keyword-driven testing are also known as script wrappers, because they wrap the technical part of the test (the actual test scripts and the test execution tool) so that the testers only need to know about the high-level keywords.

Keywords may be held in spreadsheets or tables, and longer sequences executions of test scripts can be specified by sequences of keywords.

A test sequence defined by keywords in a table may look like this:

**Ex.**

Keyword	P1	P2	P3
Create customer	Mr.	Paul	Smith
Create customer	Ms.	Anna	Philipson
Find customer	Ms.	Anna	Philipson
Edit customer	, ,Philipsson		
Find customer	Mr.	Pail	Smith
Find customer	Mr.	Paul	Smith
Delete customer	Yes		

Each keyword has a number of parameters with specific meanings. See if you can figure out what the meanings are.

Keyword-driven test is getting more and more sophisticated, introducing several levels of abstraction between the tester and the technical test scripts.

Test wrapping tools are available commercially and as open-source, but they are also very often homemade and usually quite simple, yet very effective.

Keyword-driven testing requires a good overview of the test assignment and a high level of abstraction as all parameterization does. This is demanding but can be rewarding for the test in the long run.

The *advantages* of these tools are primarily seen from the point of view of those controlling the test execution, especially if these are domain experts rather than test analysts. For test executioners it is easier to use keyword-driven testing rather than test script directly, because:

- ▶ Keywords that reflect the business can be chosen
- ▶ Test execution can be done automatically by nontechnical people based on the keyword lists
- ▶ The keyword list is robust to minor changes in the software

- The implementation of the keywords is independent of the implementation of the underlying scripts, so that the same keyword lists may be used with scripts in a number of different scripting languages being executed in different execution tools

Using keyword-driven testing does not ease the work with the actual test scripts. They still need to be established (captured or written) and maintained, and they need to be able to be executed with different parameter values.

The *pitfall* here is that extra layers are put in between the test executer and the product under testing. It requires more coordination and communication between the people involved to maintain the integrity of the layers in the testware.

Last but not least—and again a possible advantage rather than a pitfall—is the fact that keyword-driven testing requires extra care in configuration management. This kind of testing has several layers of testware instead of “just” test scripts to keep track of and to keep consistent.

### 9.3.3.7 Comparison Tools

Comparison tools are used to find differences between the expected and the actual results.

These tools range from very simple comparison facilities, like in Word, for example, to very advanced, dedicated tools. Test execution tools normally have some comparison facility included.

The tools may be able to compare, for example, values in files or on screens, bitmaps, and positions.

The *advantage* is that these tools can compare large amounts of data very fast and without getting tired.

The *pitfall* is that they may produce enormous amounts of reported data of which only a fraction is relevant. The tools can, and should, have filtering or masking possibilities (for example, to allow them to ignore dates or to ignore positions of objects and concentrate on the contents).

A comparison tool I use quite often is the window in my office. When I need to compare two texts or two drawings on paper, I place the papers on top of each other and hold them against the window. Differences are usually easy to spot.



### 9.3.3.8 Fault-Seeding and Fault-Injection Tools

These types of tools are used to support the defect-based test technique fault-injection (or fault seeding) discussed in Section 4.3.2.

The tools create or inject faults (or defects) into the software component under testing. The tools can work either on the source code, changing the code in prespecified ways, or on the compiled code, changing the structure of the code.



In both cases new versions of the component under test are created with the specified defects.

The *advantage* of the fault-seeding and fault-injection tools is that many defects may be injected in a systematic way to support these defect based techniques.

The *disadvantage* of the tools is that the defects are not necessarily realistic and may not be found by the specified tests.

### 9.3.3.9 Web Tools

These days testing never stops. With more and more Web-based products around, we need to constantly monitor that the products are doing well. Products being Web-based means that some issues are out of our hands (for example, hyperlinks and server and network availability).

Hyperlink testing tools are used to check that no broken hyperlinks are present on a Web site.

These tools often have additional functionality such as HTML validation, spelling, and availability check. The facility is often built into other tools (e.g., HTML development tools).

Monitoring tools are used for Web-based products, most typically e-commerce and e-business applications. The tools monitor the product's availability to customers and the service level (performance and resource usage). The tools will issue warnings if the monitoring shows that something is not as expected.

Many free Web tools are available for this type of tool. Many are also proprietary, and they can be quite sophisticated.

The *advantage* of these tools is that they can check all hyperlinks very quickly. It is important for the trustworthiness of a Web site that there are no broken links. Links change very quickly, and it can be quite an eye-opener to run such a check for the first time on your "perfect" Web site. The tools also give us a chance to know if things are not working as they should before the users find out.

There is *no disadvantage* using these tools. It should be an ongoing activity, at least once a day, to perform these checks.

## 9.3.4 Tools for Technical Test Analysts

### 9.3.4.1 Static Analysis Tools

Static analysis can be performed on code as well as on architecture. Static analysis is discussed in detail in Section 4.5. Most static testing is performed by people, but some types are supported by tools.

Static analysis tools examine the written code to detect, for example, variable anomalies, to check adherence to defined coding rules, and to collect measurements concerning the code, for example, cyclomatic complexity and Web site balance.

The code is not executed in static analysis, and no test cases are executed either.

One *advantage* of automated static analysis is that the tools find all occurrences of the faults they are looking for. Tools do not get tired or “blind” to faults they have seen many times before.

Static analysis requires some coding standards to check against to find deviations, and that can also be considered an advantage. The more structured and uniformly the code is written, the easier it is to maintain.

The *disadvantage* of static analysis is that some tools—especially some older tools—may find a number of “incidents” that are not faults after all. The reports for static analysis can be overwhelming with many things that can be disregarded, and that can make it difficult to find the “gold nuggets.”



#### 9.3.4.2 Dynamic Analysis Tools

Dynamic analysis tools are used to provide information about the behavior and state of software while it is being executed. These tools primarily give run-time information about memory handling and pointers.

Memory handling is concerned with allocation, usage, and deallocation of memory. The tools can detect memory leaks, where memory is gradually being filled up during extended use, long before it actually happens. Some coding languages prevent such defects from happening; others don’t, for example C and C++.

Pointers are used to handle dynamic allocation of memory and the dynamic analysis tools can identify unassigned pointers, that is, pointers pointing at “who-knows-what.” They can also detect faults in pointer arithmetic.

The *advantage* of these tools is that they can find faults that are almost impossible or very expensive to find in other ways. They don’t need specific test cases, because they report on what is going on while other test cases or scenarios are executed.

A *disadvantage* is that the code is instrumented by the tool in order for the tool to catch the run-time information. This means that it is not strictly the “real” code we are testing. It can also have an adverse impact on performance, and that can pose problems if we are testing real-time software.

One thing to be aware of is that different dynamic test tools may report different types of problems because of the way they are implemented.

A special type of dynamic analysis tool is *coverage measurement tools* or analysis tools. These tools provide objective measurement for some structural or white-box test coverage metrics, for example,



- ▶ Statement coverage
- ▶ Branch coverage

The *advantage* of these tools is that objective measurements to be used in the checking against test completion criteria are delivered in a fast and reliable way.

Some tools can also deliver reports about uncovered areas. The more fancy ones produce colored reports where covered code is shown in one color and uncovered code in another. This is a great help when more test cases must be designed to obtain a higher coverage.

The *disadvantage* is that the code is instrumented and that the tools log information during execution. This may affect the performance, and it can be a problem for real-time systems.

#### 9.3.4.3 Performance Testing Tools

Performance testing tools are used to:

- ▶ Generate large volumes or loads on the product
- ▶ Measure the performance of the product under the controlled circumstances

The tools can be used to create the volumes specified in the volume requirements and necessary for volume testing. This may be the number of concurrent users, the amount of memory to be used, the number of information items of a given type (e.g., customers or patients), or the number of transactions per time unit.

The usage of the tools for stress testing is similar to the one described for volume testing.

For performance testing the tools can be used to measure what the performance is under given circumstances.

The performance testing tools can provide very useful reports based on collected information, often in graphical form.

The *advantage* of these tools is that they can provide information about “bottleneck” areas relatively inexpensively before the product hits the real world.

There are *no disadvantages* of these tools, and hence no excuse for not using them. All too many products have no or insufficient performance requirements and turn out to be unable to cope with real-life volumes and loads. It is much better to get these aspects tested before the product breaks down when the first set of users starts using it.

#### 9.3.5 Tools for Programmers

##### 9.3.5.1 Debugging Tools

Debugging tools are NOT testing tools!

They are related to testing, since they are used by programmers to pinpoint defects. For this purpose they are a very efficient aid.



Debuggers allow programmers to:

- ▶ Execute the code line by line
- ▶ Insert break points
- ▶ Control and set values of variables at break points

Note that when testers use debugging tools to locate defects in testing tools, test scripts, or other types of testware, they do not do that in their capacity of testers, but as developers of testware. This may seem like quibbling, but for common understanding and communication purposes it is important to be able to distinguish between different roles, even when they are filled by the same person.



The *advantage* of these tools is that they can save the programmers a lot of time during detailed fault hunting. It can also be motivating for some testers with a development background to work with the programmers and use these tools to pinpoint not only the failure, but also the fault.

On the other hand the *pitfall* is that programmers can waste a lot of time if the tools are used in an undisciplined way or to play with.

## Questions

1. What are the activities in the tool acquisition process?
2. What is a “fool with a tool,” and what does that mean?
3. Who should be on a tool selection team?
4. What is a testing tool strategy used for?
5. What should be considered in a tool acquisition business case?
6. What is the most important type of requirement for a tool?
7. What are the ways in which a tool can come into existence in an organization?
8. What should be investigated about the supplier?
9. What could an evaluation scale be like?
10. What is important in a trial?
11. What are the activities in tool implementation?
12. What are the roles in tool introduction?
13. What are the goals of a pilot project?
14. What is the most important part of the actual rollout?
15. What must be done with tools used in testing?
16. How can testing tools be categorized?
17. What is the categorization of testing tools used in this book?
18. Which tool category does not contain testing tools?
19. What do we need to be careful about for test management tools?
20. What activities do configuration management tools support?
21. What is the pitfall of these tools?
22. What does a test design tool do?

23. What are data preparation tools used for?
24. What is an oracle in testing context?
25. When should simulation tools be used?
26. What are test harness and drivers used for?
27. What are test execution tools also called?
28. What do test execution tools use for execution?
29. What are the advantages and pitfalls of test execution tools?
30. What is keyword-driven testing?
31. What are tools for keyword-driven test also called, and why?
32. What gets even more important when keyword-driven testing is used?
33. Where are comparison tools often found?
34. What is the disadvantage of fault injection tools?
35. What can Web tools do?
36. What are the advantages of static analysis tools?
37. What do dynamic analysis tools provide information about?
38. What can some advanced coverage measurement tools provide?
39. What can performance testing tools do?
40. What is a debugging tool?

## Appendix 9A List of Testing Tools

This list was found on the Web site:

[www.aptest.com/resources.html](http://www.aptest.com/resources.html) on August 19, 2007.

Source Testing Tools			
AdaTEST	Automate!Test	TestComplete	
AQtime	Manager	TestWorks	
BoundsChecker	Automated Test	Unified Test Pro	
Bullseye Coverage	Designer	Vermont HighTest Plus	
CMT++	AutoTester One	VNCRobot	
Code Coverage	Avignon Acceptance	WinRunner	
CodeCheck	Testing System	X-Unity	
CodeWizard	BugHuntress		
CTA++, CTB	CAPBAK/X, CAPBAK/	<b>Performance Testing Tools</b>	
CTC++	MSW	BugTimer	
devAdvantage	Certify	DB Stress	
Diversity Analyzer	CitraTest	LoaddeaTest	
GlowCode	Code Testing Tool Pro	LoadRunner	
Insure++	Eggplant	Monitor Master	
LDRA Testbed	Eventcoder	IxLoad	
Leak Check	FERRET	QACenter Performance	
Logiscope	GUITAR	Edition	
OSPC	Haven	Scapa StressTest for	
Panorama	Holodeck	Citrix MetaFrame	
McCabe TQ	JPdfUnit	Shunra\Storm	
PolySpace Suite	MITS.GUI	SilkPerformer	
Predictive Lite	PETA	SSW Performance PRO!	
Prevent	PyUnit	97	
Purify	QAcenter	TestLoad	
TBGEN	Replay Xcessory	Vantage	
TCAT C/C++	Repro	WinFeedback	
TCMON	SAP Software Quality	XtremeLoad	
Test Coverage	Assurance Testing Tools		
	ScriptMap for Siebel	<b>Java Testing Tools</b>	
	ScriptTech	Abbot	
<b>Functional Testing Tools</b>	Silktest	AdaptiveCells/J	
.TEST	Smalltalk Test Mentor	AgileTest	
AberroTest	Squish	Agitator	
AETG Web	TALC2000	AppPerfect DevSuite	
	TestArchitect		

Bugkilla	SQS/Test Professional	SiteTechnician
Cactus	TestIt!	Validation Spider
GJ-Coverage	TurboData	W3C Link Checker
GJTester	utPLSQL	Weblint Gateway
GUIDancer		Web Page Backward
JCover		Compatibility Viewer
JCover		Web Page Purifier
Jemmy	<b>Link and HTML Testing Tools</b>	XML Validation
JMeter	AccVerify/AccRepair	Web Functional Test
JStyle	ChangeAgent	Tools
JSystem	CSE HTML Validator	actiWATE
jtest	Cyber Spyder Link Test	Astra QuickTestTM
JUnit	Dead Links	AutoTester One
JVerify	HTML Candy	Badboy
KCC	HTML PowerTools	Canoo WebTest
LISA	HTML Tidy	eValid
Panorama	InFocus	IeUnit
Marathon	Link Checker Pro	Imprimatur
QEngine	LinkRunner	Internet Macros
qftestJUI	LinkScan	HTTP::Recorder
QStudio Enterprise	LinkSleuth	iRise Application
TCAT/Java	Link Validator	Simulator
Embedded Test Tools	MOMspider	ITP
Message Magic	Ramp Ascend	LISA
Reactis Tester	Real Validator	MaxQ
TBrun	Truwex website QA	Netvantage Functional
Tessy	tool	Tester
TestQuest Pro	WebLight	PesterCat
USBTester	WebQA	QA Wizard
VectorCAST		Ranorex
<b>Database Testing Tools</b>	<b>Online Link and HTML Testing Services</b>	Rational Robot
AETG	Audit Blossom	Sahi
Data Generator	Bobby	SAMIE
Data Generator	CSSCheck	Selenium
DataTect	CSS Validation Service	SilkTest
DTM DB Stress	Dr. Watson	SoapTest
ER/Datagen	HTML Validator	soapui
Jenny	HTML Validation Service	Solex
Jumpstart	Link Alarm	Squish
SQL DB Validator	NetMechanic	swete
SQL Profiler	Site Check	TestSmith
		TestWeb
		vTest

WatIN	Web Performance	Jitterbug
Watir	Trainer	JTrac
WebAii	Web Polygraph	Mantis
Webcorder	Web Roller	MyBugReport
WebInject	Web Server Stress Tool	Ozibug
WebKing	WebSizr	Perfect Tracker
WET	Web Performance Test	ProblemTracker
WSUnit	Services	PR Tracker
Yawet	Load Gold	QEngine
<b>Web Security Testing Tools</b>		SpeeDEV
QA Inspect	SiteStress	Squish
<b>Web Performance Testing Tools</b>		Task Complete
ANTS	webStress	teamatic
Dotcom-Monitor		TeamTrak
forecast		TrackStudio
http_load		VisionProject
Jblitz		Woodpecker IT
LoadTracer		yKAP
Microsoft Application Center Test	<b>Web-Based Bug Tracking</b>	<b>Bug Tracking Applications</b>
NeoLoad	AceProject	assyst
OpenLoad	AdminiTrack	BridgeTrak
OpenSTA	ADT Web	BugRat
Portent	Bug/Defect Tracking	BugSentry
PowerProxy	Expert	Bug Trail
Proxy Sniffer	BugAware.com	Defect Agent
PureLoad	bugcentral.com	Defect Manager
QuotiumPro	BUGtrack	Fast BugTrack
Siege	BugHost	GNATS
SilkPerformer	BugRoster	Intercept
StressIT	BugStation	IssueView
Site Tester 1.0	Bug Tracker	JIRA
TestMaker	Bug Tracker Software	ProjecTrak
WAPT	Bug Tracking	PVCS Tracker
Wbox	Bugvisor	QAW
Web Application Stress Tool	Bugzero	Support Tracker
Webload	Bugzilla	SWBTracker
WebPartner TPC	Census BugTrack	TestTrack Pro
	DefectTracker	Track
	Defectr	ZeroDefect
	DevTrack	Test Management Tools
	Dragonfly	ApTest Manager
	ExDesk	
	FogBUGZ	
	Fast BugTrack	
	Footprints	
	GranPM	
	IssueTrak	
	JIRA	

Extended Test Plan	FanfareSVT	Caliber
QADirector	Fault Factory	Gatherspace
SilkPlan Pro	InterWatch	RequisitePro
T-Plan Professional	iSoftTechTAS	SpeeDEV RM
TestDirectorTM	LANTraffic	SteelTrace
Test Manager Adaptors	Maxwell	
TestLog	NetDisturb	<b>Other Products</b>
	NetworkTester	Aprobe
<b>API Testing Tools</b>	nGenius Performance	Ascert
ADL project repository	Monitor	Bug Shot
MITS.Comm	NuStreams 2000	Exchange Simulator
DejaGNU	Silvercreek SNMP Test	Funnel IT
TET	Suite	InSpec
	SNAsim	KaNest
<b>Communications</b>	VoIP Conformance Test	LogStomper
<b>Testing Tools</b>	Suite	QAcenter 3270 Edition
AdventNet Simulator	WAN Emulator	SOAPSonar
ANVL		Stabilizer
Chariot		TestBench400
Cheetah		TestOOB
Drive Test		
Emulation Engine XT		
	<b>Requirements</b>	
	<b>Management Tools</b>	
	Analyst Pro	
	Doors	

## People Skills

People work for one single reason: We have to. There are, however, many explanations for why we have to—also some more specific to testers.

The individual's testing capability can be derived from experience and/or training in one or more of the following areas: users, development, and testing. No matter who we are, interpersonal skills such as giving and receiving criticism, influencing, and negotiation are all important in the role of testing.

People are different, and it is an advantage to have a variety of personality types within the test team. The best combinations may be ensured at times of recruiting. Even if that is not possible, knowledge of certain patterns of behavior can help us enhance the team we are working with.

Testers work in many different organizations, and they all have different organizational structures for testing and for other activities. Communication within an organization is essential, not least in testing.

### 10.1 Individual Skills

Testing is a profession. It requires certain skills and capabilities of the individual testing practitioners. It also requires people with certain human characteristics or personality types.

People have individual personalities. A person's personality is very difficult to change; we are born with many of the traits, and the rest have been chiseled into us from then on!

There are some common traits that will help testers in their position as professional testers. A tester should preferably, and very generally speaking, be:

- ▶ Intelligent—Testing is an intellectual type of work
- ▶ Creative—Testing needs to be inventive to be effective
- ▶ Persevering/enduring—Testing needs to go on and on despite resistance and pressure

### Contents

- 10.1 Individual Skills
- 10.2 Test Team Dynamics
- 10.3 Fitting Testing in an Organization
- 10.4 Motivation
- 10.5 Team Communication

- ▶ Systematic—Testing needs to have a trustworthy coverage
- ▶ Pragmatic—Testing is sampling in its nature
- ▶ A good communicator—Testing has many stakeholders
- ▶ Courageous—Testing can be perceived to bring bad news
- ▶ Mature—Testing is a demanding profession



Personality types and how they can be deployed to the advantage of test teams are discussed in more detail in the next section.

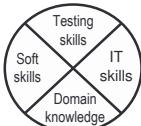
Professional testers need to have training and experience in testing. Training includes:

- ▶ Education, either from an educational institution or in the form of courses in testing theory. The testers should learn to remember and understand test-related terms, concepts, and statements.
- ▶ On-the-job training and mentoring. The testers should learn to apply their knowledge.
- ▶ Carrying out of test tasks. The testers should learn to see structures and principles and divide the task into smaller tasks.
- ▶ Experience exchange and further education. The testers should learn to combine and think in abstract terms.

Even though testing is a specific profession many testers have had other careers before they became testers. I have encountered a former dentist, a lawyer, several chemists, and many, many others. However, most testers, who have not started out as testers, either have a background in development or in the domain of the product they are working with.

Some say that a tester must have a development background. It is not necessarily so, but it certainly helps. Knowledge of how software development is done gives an invaluable insight into what could cause errors to be made and how faults can possibly be introduced.

Having a background in the product domain can also be a great help in a testing career. It facilitates the understanding of the requirements and the necessary test environment. Knowledge of the domain creates valuable credibility from the point of view of the users of the product. In fact it can be a good investment to provide testers without relevant domain knowledge with a feeling for where the product they are working on is going when it is released.



Many testers in the economic sector have a background in banking.

### 10.1.1 Test Roles and Specific Skills

Testing is not just one single activity. If we take a close look at the test process and all the tasks testers have to perform, we can see that there are a number of different test roles involved.

The test roles may be defined as:

- ▶ Test leader (manager or responsible)
- ▶ Test analyst/designer
- ▶ Test executor
- ▶ Reviewer/inspector
- ▶ Domain expert (user representative)
- ▶ Test environment responsible
- ▶ (Test)tool responsible



Each of these roles requires specific skills and capabilities of the people filling them, apart from the general traits and skills required for all testers. The following table lists the most important ones:

	<b>Should have training and/or experience in</b>
<b>Test leader</b> 	Test policy and strategy for the organization Testing standards Test management: estimation, planning, monitoring, control, reporting, managing people
<b>Test analyst/designer</b>	Analysis of requirements and other specifications Design of test cases Effective usage of test case design techniques Building and documenting test procedures 
<b>Test executor</b>	Executing, recording, and checking tests 
<b>Reviewer/inspector</b>	Static test techniques 
<b>Domain expert</b>	Basic test principles
<b>Test environment responsible</b>	Platform(s) Test database administration
<b>Test tool responsible</b>	Platform(s) Specific tool(s) 

### 10.1.2 Testing by Other Professionals

Even though testing is a profession in its own right, people in other professions can also participate and contribute in the performance of the testing activities.

Users may be good testers. This applies to both future users of the new product and actual users of similar products or earlier versions of the product. The users obviously see the product from the users' perspectives. In other words users have ideas about the future use of the product. This provides insight into where failures would have the greatest impact (i.e., it provides input to the risk management).

Users are best involved in requirements analysis—this is where it all starts. But they can also be very useful in the test, even if in-depth knowledge of the domain does not reside with (naïve) users.

Developers may be good testers. They know how difficult requirements analysis is, how difficult design is, and how difficult coding is, and this means that they have insight into where errors may have been made, and hence where the faults may be.

Developers are best involved in static testing. They can also contribute in dynamic testing, not least in component testing and component integration testing.

### 10.1.3 Interpersonal Skills

Interpersonal skills are important in the roles of testing. Testers need to know how to:

- ▶ Give and receive criticism
- ▶ Influence people
- ▶ Negotiate



*Giving criticism* is very difficult for most people. We don't want to hurt each other—most people prefer to have peace rather than pointing out failures.

Again we need to consider what testing is. It may be seen as a destructive activity, since the tester has to find and register failures. However, registration of failures is not criticizing the developer. Testing is a very constructive development activity—it contributes to the possibility of everybody reaching the common goal of delivering a product of a good quality, learn, and becoming even better in the future.

Testers do however have to give criticism from time to time. The basic rules for doing that are:

- ▶ Stay calm

T: "Is it convenient for you if we talk about my findings now?"

Ex.

- ▶ Keep to the facts

T: "During the last hour I think I have come across 27 failures in the customer invoicing feature."

Ex.

- ▶ Don't blame

T: "We need to figure out how this situation can be stabilized—is there anything I can do?"

Ex.

- ▶ Keep an open mind—you could be wrong!

*Receiving criticism* is just as difficult as giving it. We feel threatened, and some of our needs (respect, recognition, and security) seem to be jeopardized.

Testers do however have to take criticism from time to time if we want to get better at our job. The basic rules for doing that are:

- ▶ Listen carefully for the tiniest bit of truth

D: "You keep disturbing me with all those \* failures!"

T: "Yah, I'm sorry I come barging in here every 10 minutes."



- ▶ Ask for clarification of view and goals

T: "How do you suggest I go about reporting the failures?"

- ▶ Make concessions—when the criticism is legitimate, you have to admit to it frankly—otherwise you'll lose credibility!

*Testing interacts* with many other development and supporting activities. We are dependent on other people's decisions and other people's schedules.

Often testers harbor a feeling of being victims. We find ourselves in impossible situations, and we don't do much about it because: "This is just the way things are!" There is no reason for bending under this "law of necessity." In fact testers can usually influence more than they think—especially if we start early.

Some of the areas, where testers are dependent on others, but where we may also use our influence, are:



- ▶ Delivery order—By asking and explaining why
- ▶ Delivery quality—By defining entry criteria
- ▶ Delivery date—By negotiating with other managers
- ▶ Planning of test activities—By talking to other managers
- ▶ Allocation of resources—By negotiation with management
- ▶ Classification of failures—By using a scheme
- ▶ Training—By asking and participating
- ▶ Process improvement—By contributing and participating

When we assert our influence on other people it is useful to remember that demands create resentment, while requests for help usually create kindheartedness.

**Ex.**

T: "It would be a great help for us if we could start testing the components xx and yy first. Would you be able to help us do that?"

Like in all other communication we need to listen first and then talk. When we listen we must listen for reasons, goals, fears, and threats from the other party's point of view. And when we talk, we most of all have to explain. We have to explain our reasons, goals, fears, and threats.

Most of the time we can come to an agreement simply by talking and explaining our needs and constraints to each other.

However, sometimes we get into a situation where we have to *negotiate*, that is, to engage in bargaining to reach agreement.



A few basic rules about negotiations are:

- ▶ Look behind the positions to the real interests
- ▶ Work with BATNAs—Best alternative to a negotiated agreement
- ▶ Walk away if negotiation is going nowhere
- ▶ Identify options as parts of the solution
- ▶ Go for a win-win solution
- ▶ Aim at an atmosphere of common problem solution



## 10.2 Test Team Dynamics

Imagine if everybody were like you...

Would life be better or worse for that?

People have different personalities. This has been known since the ancient Greek philosophers defined four temperaments:

- ▶ Phlegmatic
- ▶ Sanguine
- ▶ Choleric
- ▶ Melancholic

The philosophers also said: “We all have our share of each—in different mixtures.”

Others have studied personalities including Freud, Jung, and Myers-Briggs. Based on Jung’s work, Myers-Briggs defines 16 personality types composed from four dimensions. The dimensions are:

- ▶ How do you get energy:  
Extraversion (E) / Introversion (I)
- ▶ How do you collect information and knowledge:  
Sensing (S) / Intuition (N)
- ▶ How do you decide:  
Thinking (T) / Feeling (F)
- ▶ How do you act:  
Judging (J) / Perceptive (P)



The Greek view is quite simple; the Myers-Briggs view rather complex; and they are both concerned with the individual person as just that: an individual.

### 10.2.1 Team Roles

Dr. Meredith Belbin and his team of researchers based at Henley Management College, England, have studied the behavior of managers from all over the world during a period of over nine years. Their different core personality traits, intellectual styles, and behaviors were assessed during the exercise.

Results from this research showed that there are a finite number of behaviors or team roles. A team role as defined by Dr. M. Belbin is: “A tendency to behave, contribute and interrelate with others in a particular way.”



Belbin has defined nine team roles based on his studies. They each describe a pattern of behavior that characterizes a person’s behavior in relationship to others in a team.

The nine team roles are divided into three role types to create an overview and a deeper understanding of how the roles work.

The Belbin roles are:



- ▶ **Action-oriented**
  - ▶ Shaper
  - ▶ Implementer
  - ▶ Completer
- ▶ **People-oriented**
  - ▶ Coordinator
  - ▶ Team worker
  - ▶ Resource investigator



► **Cerebral**

- Plant
- Monitor
- Specialist

All of the roles have some valuable *contributions* to the progress of the team in which they act. They also have some *weaknesses* that may have an adverse effect on the team.

The contributions and weaknesses are summarized in the following table.

<b>Team Role</b>	<b>Contributions</b>	<b>Weaknesses</b>
<b>Shaper</b>	Challenging, dynamic, thrives on pressure. The drive and courage to overcome obstacles.	Prone to provocation. Offends people's feelings.
<b>Implementer</b>	Disciplined, reliable, conservative, and efficient. Turns ideas into practical actions.	Somewhat inflexible. Slow to respond to new possibilities.
<b>Completer/finisher</b>	Painstaking, conscientious, anxious. Searches out errors and omissions. Delivers on time.	Inclined to worry unduly. Reluctant to delegate.
<b>Coordinator</b>	Mature, confident, a good chair-person. Clarifies goals, promotes decision-making, delegates well.	Can often be seen as manipulative. Off-loads personal work.
<b>Team worker</b>	Cooperative, mild, perceptive, and diplomatic. Listens, builds, averts friction.	Indecisive in crunch situations.
<b>Resource investigator</b>	Extrovert, enthusiastic, communicative. Explores opportunities. Develops contacts.	Overly optimistic. Loses interest once initial enthusiasm has passed.
<b>Plant</b>	Creative, imaginative, unorthodox. Solves difficult problems.	Ignores incidentals. Too preoccupied to communicate effectively.
<b>Monitor/evaluator</b>	Sober, strategic, and discerning. Sees all options. Judges accurately.	Lacks drive and ability to inspire others.
<b>Specialist</b>	Single-minded, self-starting, dedicated. Provides knowledge and skills in rare supply.	Contributes only on a narrow front. Dwells on technicalities.

Everybody is a mixture of more team roles, usually with one being dominant. An analysis of one's Belbin team role will give a team role profile showing the weight of each role in one's personality.

### 10.2.2 Forming Testing Teams

It is the test manager's responsibility to get the test team to work for a specific testing task. And it is the higher management's responsibility to choose a test manager with the right traits, skills, and capabilities.

There are two aspects to a team: the people and the roles assigned to the people. Each individual person in a team has his or her personal team role profile (for example, according to Belbin) and a number of skills and capabilities. Each role has certain requirements toward the person or the people who are going to fill it. Apart from all that, the people in the team need to be able to work together and not have too many personality conflicts.

It can be quite a puzzle to form a synthesis of all this. But the idea is to choose people to match the requirements of the roles and to fit together as a team.

The ideal situation is of course when the manager can analyze the roles he or she has to find people for and then hire exactly the right people. Advertisements and other recruitment efforts can be tailored to the needs. The applicants can be tested, both for their personal traits and for their skills and capabilities. The team can then be formed by the most suitable people—and ahead we go.

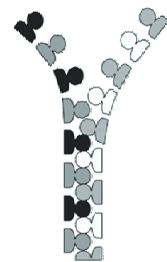
Unfortunately life is rarely that easy, although it sometimes is. In most cases either the test manager has an already defined group of people from which to form a team, or he or she has a limited and specific group of people to choose from. It can also be that the manager has to find one or more new people to fill vacancies on an existing team.

In all cases the knowledge of people's Belbin team role profiles is a great advantage. Even people in teams that have worked together for a long time can benefit from knowing their own and the other team members' team role profiles.

I once worked on a team with many frictions and mistrust. One of the team members had heard of the Belbin roles and we all had a test. That was a true revelation to us all. The two team members with the most friction between them were very different types. They had both been completely at a loss as to why the other acted as he did. Having understood that it was not ill will, but simply a question of being very different personalities, they worked much better together in the team.

The contributions as well as the weaknesses of each team role must be considered. A well-formed team is a strong team, and a team tailored for the task is the strongest team you can get.

Forming teams and getting them to work is not an easy task. There is no absolute solution.



## 10.3 Fitting Testing in an Organization

### 10.3.1 Organizational Anchorage

Testing is always done in an organization, and it can be anchored in many different ways in an organization. It can in fact be anchored in several places during the course of a development project and the subsequent maintenance period for the product running in production.

Let's first take a look at the different organizational units involved in testing. They can, for example, be:



- ▶ Product management
- ▶ Project management
- ▶ Quality assurance department
- ▶ Development department
- ▶ Development team
- ▶ Internal test department or test team
- ▶ External test organization
- ▶ Internal or external consultants
- ▶ Sales/marketing department
- ▶ Support organization
- ▶ Internal IT-department
- ▶ The customer
- ▶ Present and future end users
- ▶ Subcontractor(s)
- ▶ Process or method department

Distributing the responsibility for all the testing activities for the appropriate testing levels and the defined testing roles over organizational units is a three-dimensional jigsaw.

A number of rules should be observed for this puzzle:

- ▶ Testing requires one or more test teams—We can for example have a test team for component and integration testing, and another team for system testing.
- ▶ Test teams are composed of a number of roles—All the roles must be covered for the entire test task for a project, but it could be that the component testing does not require a test environment responsible or a domain expert.
- ▶ A role can be filled by one or more people—This depends on the size of the team. We may for example need one responsible for the test, a number of test designers, and an even greater number of test executors for a large test task.
- ▶ One person can fill one or more roles—Again this depends on the size.

The test designer can for example also be the test executor. Here it is important to remember that less than 25% assignment to a role = 0 (i.e., don't cut your slices too thinly).

- ▶ People may come from different organizations—The developers could be test designers and executors for the component testing; people from an independent test department could fill these roles for system testing; and customer representatives could fill them for acceptance testing.

The distribution of the roles must be done with great care and documented explicitly and precisely in the test plan and/or other relevant plans.



### 10.3.2 Independence in Testing

Testing should be as objective as possible. The closer the tester is to the producer of the test object, the more difficult it is to be objective.

Producers usually find it quite difficult to try to get their own products to fail. They have already done their best, so how can there be faults left in the product. Impossible! Furthermore, producers carry with them to the testing any assumptions on which the production was based. We therefore don't get a new viewpoint in the object.

Identical considerations apply for the project team testing each other's products, even though to a smaller degree.

The concept of independence in testing has therefore been introduced. The degree or level of independence increases with the "distance" between the producer and the tester. Six levels have been defined:

1. Producer tests his or her own product
2. Tests are designed by a different person than the producer, but one with the same responsibilities, typically another developer
3. Tests are designed by a tester who is a member of the same organizational unit as the producer reporting to the same boss
4. Tests are designed by testers independent of the producing organizational unit, though still in-house
5. Tests are designed by testers belonging to an external organization working in the production organization (consultants)
6. Tests are designed by testers in an external organization (third-party testing)



As can be seen in the list, the point is who designs the test cases. In structured testing the execution must follow the specification strictly, so the degree of independence is not affected by who is executing the test. In less scripted tests, like exploratory testing, the independence is between the

producer and the test executor.

The strategy must determine the necessary degree of independence for the test at hand. The higher the risk, the higher should the degree of independence be.

The independence usually varies for the different levels of testing. In component testing we often see the lowest level of independence (= no independence) even though the same concept in reviews does not seem acceptable. The higher the test level the higher the independence usually is.

The three highest levels of independence include crossings of organizational borders. We have specific names for these types of tests, namely:



- ▶ Distributed testing—The test is carried out by people belonging to the same organization, but distributed geographically (or organizationally)
- ▶ In-sourced testing—The test is carried out in the development organization, but by people reporting to a different organization (consultants)
- ▶ Out-sourced testing—The test is carried out in a different organization by this organization's own people

These ways of distributing responsibilities have advantages and disadvantages, and they pose specific requirements on the organizations.

The obvious advantage is the inherent independence of testing. Other advantages may be lower wages and overcoming shortage of staff.

The disadvantages will have to be taken into account in the risk analysis for the testing project, and mitigations must be planned. People not having taken close part in the development will be less prepared for the testing task and might take longer to produce test specifications.

The risks related to distributed, in-sourced, and outsourced testing fall within the areas of:



- ▶ Process descriptions
- ▶ Distribution of work
- ▶ Quality of work
- ▶ Culture
- ▶ Trust

The involved organizations must be aware of their own processes and the processes to which the other parties work. If processes cannot be shared the interfaces between them must be made very clear.



The work breakdown structure for the task assignment must be performed to a rather large level of detail, and all involved must agree. The most important part is for them to agree on the distribution of the tasks, so that nothing is left out and nothing performed twice.

Testing work to be done by people not having been closely involved in the development of a product requires precise and comprehensive basis documentation. A test team sitting far away from the development team cannot easily ask what to expect when documentation is missing or how to interpret unclear documentation. This is perhaps a blessing in disguise: All other things equal, the better basis documentation, the better the product will be.

For outsourced testing the quality assurance of the work being done is a specific risk. We test work by testing its results, but we cannot test the testing as such. We need to be prepared to do a thorough review of work products produced by the outsourced organization and require comprehensive documentation of fulfilled completion criteria.

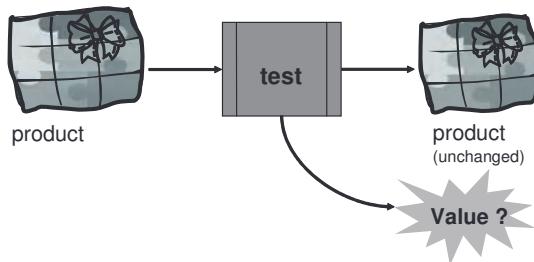
Without trust between the people working together and relying on each other the work will not be done properly. This goes for all kinds of work, but especially when work is split up between different organizations. Mutual trust must be the starting point, but all parties should remember that trust is very easily lost and hard to earn back.



## 10.4 Motivation

Why do we work? And why do we test?

On the surface, testing does not bring about any value—the object under testing is in principle unchanged by the test.



So is it really worth it—for the company and for us?

We have previously seen how testing brings value to the company by the way the information we collect during the testing is used. This is why our company pays us. But do we work just for money?

Traditionally a company or organization has been regarded as a control system. The employees were a variable cost factor to be minimized through rationalization. The salary was the main purpose for the employees and a tool for the management. Today we see and know that at least in the western world the salary is far from the only motivation.

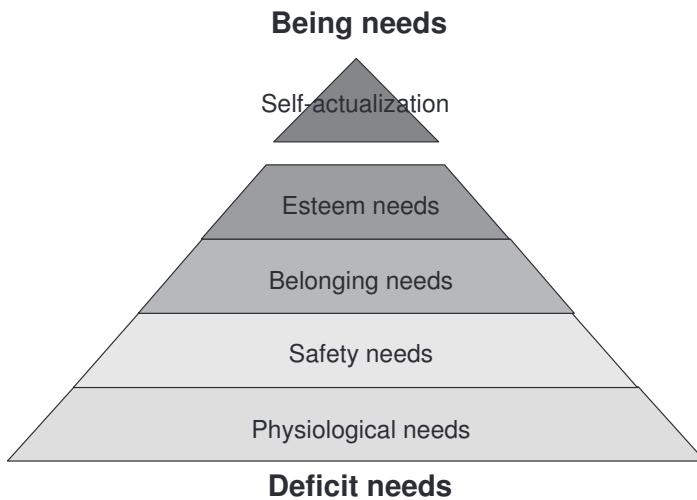
A number of American scientists, Maslow, McGregor, Herzberg, and Hackman, as well as Danish professor K.B. Madsen, have contributed to the understanding of why we work.

### 10.4.1 Maslow's Pyramid of Needs



Abraham Maslow, 1908–1970, had a Ph.D. in psychology from the University of Wisconsin. One of the many interesting things he noticed while working with monkeys early in his career was that some needs take precedence over others. For example, if you are hungry and thirsty, you will tend to try to take care of the thirst first. After all, you can do without food for weeks, but you can only do without water for a couple of days! Thirst is a “stronger” need than hunger. Likewise, if you are very, very thirsty, but someone has put a choke hold on you and you can’t breathe, which is more important? The need to breathe, of course.

Based on these observations, Maslow created his now-famous *hierarchy of needs*.



Maslow laid out five broader layers: the physiological needs, the needs for safety and security, the needs for love and belonging, the needs for esteem, and the need to actualize the self, in that order.

1. *The physiological needs.* These include the needs we have for oxygen, water, minerals, and vitamins, as well as to be active, to rest, to sleep, to get rid of wastes, and to avoid pain.
2. *The safety and security needs.* When the physiological needs are largely taken care of, this second layer of needs comes into play. You will become increasingly interested in finding safe circumstances, stability, and protection.
3. *The love and belonging needs.* When physiological needs and safety needs are, by and large, taken care of, a third layer starts to show up. You

begin to feel the need for friends, a sweetheart, children, affectionate relationships in general, and even a sense of community.

4. *The esteem needs.* Next, we begin to look for self-esteem. Maslow noted two versions of esteem needs, a lower one and a higher one. The lower one is the need for the respect of others, the need for status, fame, glory, recognition, even dominance. The higher form involves the need for self-respect, including such feelings as confidence, competence, achievement, mastery, independence, and freedom. Note that this is the “higher” form because, unlike the respect of others, *once you have self-respect, it’s a lot harder to lose!*



All of these four levels Maslow calls *deficit needs*, or *D-needs*. If you don’t have enough of something (i.e., if you have a deficit), you feel the need. But if you get all you need, you feel nothing at all! In other words, they cease to be motivating.

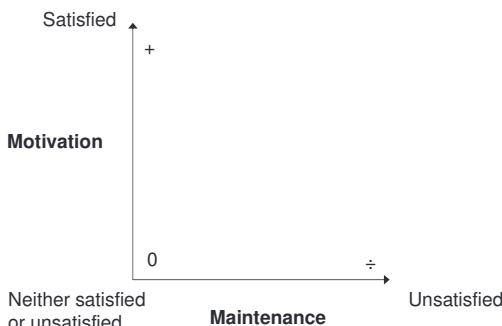
In the western world the basic needs are covered for almost everyone. Most people therefore start a couple of layers up when they are to move upwards in Maslow’s pyramid. In the IT business where people usually have higher education and higher salaries it is particularly the top two or tree layers in the pyramid that can be unsatisfied. We need to keep this in mind when we discuss motivation of testers.



#### 10.4.2 Herzberg's Factors

In the late 1950s Frederick Herzberg’s wrote the book *The Motivation to Work*. This has become one of the most replicated studies in the field of workplace psychology.

On the basis of interviews with engineers, politicians, scientists, accountants, officers, and others, Herzberg concludes that there are two factors in working situations: maintenance factors and motivation factors. The factors giving satisfaction and motivation are separate and different from the factors that create dissatisfaction. Satisfaction and dissatisfaction are not directly each other’s contrast as illustrated here:



- ▶ The opposite of satisfaction with the work is not dissatisfaction, but rather no satisfaction.
- ▶ The opposite of dissatisfaction is not satisfaction, but rather no dissatisfaction.



Motivation factors are the factors that stimulate a need for personal development, and they are embedded in the work, like performance, recognition, responsibility, and promotion. Maintenance factors include company policies and administration, relations to colleagues, salary, status, and security. Herzberg's main point is that only the work itself can give lasting motivation.

If we compare Maslow's pyramid with Herzberg's factors, we see that the motivation factors belong in the top two layers of the pyramid and the maintenance factors in the three lower.

#### **10.4.3 K. B. Madsen's Motivation Theory**

Both Maslow's and Herzberg's theories have been criticized: Maslow for not having described a universal motivational process, but one reflecting the American middle class; Herzberg for his interview form: If you ask people about good and bad aspects of their working life, they are inclined to attribute the bad ones to others, and the good ones to their own accomplishments.

Danish professor K.B. Madsen has provided a synthesis of a number of motivation theories. He sees us as being driven by internal forces (needs = motives) and controlled by external forces (incentives).

Incentives can be split into primary and secondary. Primary incentives activate our innate reactions. Secondary incentives activate our acquired reactions. Some of the more important incentives are listed here:

	<b>Innate</b>	<b>Acquired</b>
<b>Want to have</b>	Food and drink Comfort Service Security	Money Recognition Honor and praise Respect
<b>Want to avoid</b>	Pain Punishment Aggression Humiliation	Sarcasm Danger signals Frustration Verbal threats

The motives are recognizable from Maslow. They can work together, or they can be in conflict. We can be motivated by social motives like collecting and presenting information, but that is usually not combinable with activity motives like movement and suspension.

In connection with our working life K.B. Madsen summarizes his and other's research in the following statements:

*The most effective form of motivation is: interest in the task.*

Almost as effective is: reward and praise.

Least effective form of motivation is: punishment and blame.



#### 10.4.4 Testers' Motivation

Motivation theories are, in general, not particularly aimed at testers. However, we can map the general motives to the testing world.

Testers are motivated by an interest in the task. This means that testers must think or feel that it is fun to find failures to be motivated to do their job. They must see it as an intellectual challenge to reveal as many faults as possible in the given time. This is in many ways in contradiction with "acceptable, nice behavior" and requires a very specific mind-set. Testers must sometimes work hard to convince both ourselves and others around us that we work together to produce a more reliable product—not to destroy it.

Testers are motivated by reward and praise. We can also say that testers must get recognition and respect—we certainly deserve it. This can be done by management being very aware of the value testers add to the project and hence to the company. Remember what we create of value? Project information! The better we are at presenting the information and assisting the recipients in using it, the more recognition and respect we get.

The earlier the testers are involved in the project the more we can prove our value. Testers involved in early static testing can be a great eye-opener and motivator for everybody.

Testers are demotivated by punishment and blame. One (rather sophisticated) form of punishment can be lack of career path. It can be very demotivating if the organization ignores the testers and doesn't provide us with a way to climb up the organization ladder. On a more day-to-day scale lack of understanding of value of testing ("Oh, you are just overhead") or direct blame ("Stop finding (read: producing) all those failures!") can make the working life of testers very uphill.



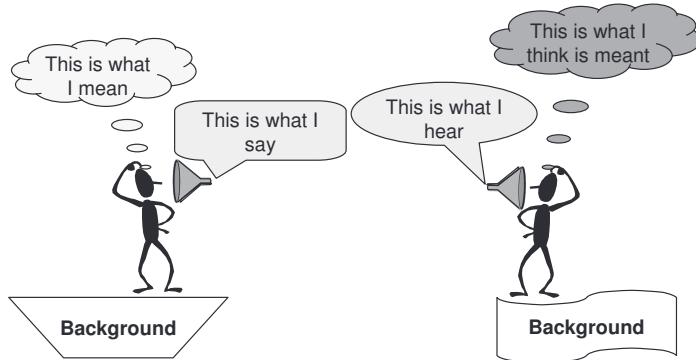
## 10.5 Team Communication

Testers need to communicate with people in all the organizational units involved in the testing. This means that testers have to communicate with many different people at many different organizational levels internally in their organization and externally, for example, with customers.

The first thing to remember about *communication is, that it is difficult*. The way of communicating where people understand each other absolutely perfectly and completely is not possible.



We are people and we are all “colored” by our personalities, our upbringing, and our experience—just to mention a few factors.



However, we have to communicate, and we have to make an effort to make it work. One of the things we can do is to understand the type of people we are dealing with, both in terms of their personality types, their education and background, and their responsibilities and working conditions.

Another thing is to accept that it is the “sender” of the information who is responsible for making it understandable to the audience. We as testers must learn that other teams talk in different languages: Management speaks “money”; users (and marketing and sales) speak “functionality and quality”; developers speak “technique.” All our communication must be targeted to the audience.

The most important and perhaps the most difficult communication lines are those between the testers and:

- ▶ Project management
- ▶ Developers
- ▶ Users



Communication with project management is most often done by the test manager. Project management needs to inform testers about aspects like expectations, resources, constraints, quality criteria, and changes in plans and discuss these factors. Testers on their part must inform project management about issues like the progress of testing and quality of the product under testing. This communication is usually based on written documentation like test strategy, test plans, test progress reports, and test summary reports but should also be accompanied by verbal communication.

The communication with development is usually done by test analysts and designers. Development needs to inform testing about issues like especially

complex or difficult areas of the product, which areas are new development, and which have “just” been updated, other areas that need special attention for various reasons, changes in requirements and/or design, changes in delivery schedule, specific difficulties during development, problems reproducing reported failures, and when and why new test objects are delivered for testing.

Testers need to inform development about failures found, problems arising during confirmation testing corrected defects, and problems concerning the number and/or types of failures.

These issues can be delicate to talk about. The information should be conveyed diplomatically as a means of improving the quality of the product—not as blame! Also here written documentation may constitute some of the communication, but verbal communication must not be eliminated completely.

The users may communicate with many different test roles depending on the organizational structure. Useful information can be provided by users to testers, for example, concerning their expectations regarding the new product, risk areas to the business in the product, assessment of the effect of identified risks, important areas in the product seen from the users' perspective, and background information about the business and the business processes.

This information can be used to support the risk analysis and the prioritization of the testing. Users will often receive or see test results. Users with little knowledge of testing may need help in the interpretation of these results.



## Questions

1. What traits are useful for a tester to have?
2. What are the seven test roles that can be defined?
3. Which one do you prefer and why?
4. What should you remember when giving criticism?
5. What should you remember when receiving criticism?
6. What are the areas where testers are dependent on other people?
7. What are the negotiation rules?
8. What are the four temperaments?
9. What are the four dimensions that Myers-Briggs defined?
10. What are the nine Belbin team roles?
11. How should test teams be formed?
12. Which organizational unit may be involved in testing, and how?
13. What are the levels of test independence?
14. What are the differences between distributed testing, in-sourced testing, and outsourced testing?
15. What are the risk areas for these kinds of test organizations?
16. Why do you work in testing? (or in other areas as the case might be)

17. What is the idea in Maslow's hierarchy of needs?
18. What is special for self-respect compared to the other needs?
19. What are Herzberg's two types of factors?
20. What is the idea in K.B. Madsen's motivation theory?
21. What is the most effective form of motivation?
22. Why is the "perfect" communication impossible?
23. Who do testers communicate with?
24. What do we communicate about?

# **Selected Bibliography**

## **Books**

- Beizer, B., *Black Box Testing*, Wiley, 1995.
- Beizer, B., *Software Testing Techniques*, Wiley, 1990.
- Black, R., *Managing the Testing Process*, Microsoft Press, 1999.
- Buwalda, H., Janssen, D., and Pinkster, I., *Integrated Test Design and Automation*, Addison-Wesley, 2001.
- Copeland, L., *A Practitioner's Guide to Software Test Design*, Artech House, 2003.
- Fewster, M., and Graham, D., *Software Test Automation*, Addison Wesley, 1999.
- Gerrard, P., and Thompson, E., *Risk-Based E-Business Testing*, Artech House, 2002.
- Gilb, T., *Software Inspection*, Addison-Wesley, 1993.
- Hass, A. M. J., *Configuration Management Principles and Practice*, Addison-Wesley, 2003.
- Hass, A. M. J., *Requirements Development and Management*, DF-17; Copenhagen, 2003.
- International Software Testing Qualifications Board, *Certified Tester, Advanced Level Syllabus, Version 2007*.
- Kaner, C., Bach, J., and Pettichord, B., *Lessons Learned in Software Testing*, Wiley, 2002.
- Kit, E., *Software Testing in the Real World*, Addison-Wesley, 1995.
- Koomen, T., and Pol, M., *Test Process Improvement (TPI)*, Addison-Wesley, 1999.
- Myers, G., *The Art of Software Testing*, Wiley Interscience, 2004.
- Perry, W. E., and Rice, R. W., *Surviving the Top Ten Challenges of Software Testing*, Dorset House, 1997.
- Pinkster, I., et al., *Successful Test Management*, Springer, 2004.
- Pol, M., Teunissen, R., and Veenendaal, E. van, *Software Testing: A Guide to the TMap Approach*, Addison-Wesley, 2002.
- Tobar, Ltd., *Incredible Visual Illusions*, Arcturus Publishing Limited, 2005.
- Veenendaal, E. van., *The Testing Practitioner*, UTN, 2002.
- Vinter, O., *The prevention of Errors through error experience-driven test efforts*, DELTA rapport D-259.
- Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.

## **Standards**

- BS 7925-2 (1998) "Software Component Testing"
- IEEE Std 829—1998/2005, "Standard for Software Test Documentation"  
(currently under revision)

IEEE Std 1028—1997, “Standard for Software Reviews”  
IEEE Std. 1044—1993, “Standards Classification for Software Incidents”  
IEEE Std. 1044.1—1995, “Guide to Classification for Software Incidents”  
ISO/IEC 9126—1:2001, “Software Engineering—Software Product Quality”  
ISTQB “Glossary of Terms Used in Software Testing,” Version 2.0, 2007

## Web Sites

<http://www.sei.cmu.edu/cmmi/products/models.html> or  
<http://www.sei.cmu.edu/cmmi/general/genl.html> (a very comprehensive description of CMMI®)  
<https://www.cis.strath.ac.uk/teaching/ug/classes/52.429/lecture10.pdf>  
[www.belbin.com](http://www.belbin.com)  
[www.iese.fhg.de/network/ISERN/pub/technical\\_reports/isern-98-32.pdf](http://www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-98-32.pdf)  
(An Encompassing Life-Cycle Centric Survey of Software Inspection,  
ISERN-98-32)  
[www.jamesmartin.com](http://www.jamesmartin.com)  
[www.sei.cmu.edu/cmm/obtain.cmm.html](http://www.sei.cmu.edu/cmm/obtain.cmm.html) (CMM® Version 1.1)  
[www.sei.cmu.edu/sei-home.html](http://www.sei.cmu.edu/sei-home.html)  
[www.sei.cmu.edu/str/descriptions/cyclomatic\\_body.html](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html)  
[www.ship.edu/~cgboeree/maslow.html](http://www.ship.edu/~cgboeree/maslow.html)  
[www.stsc.hill.af.mil/SWTesting/index.html](http://www.stsc.hill.af.mil/SWTesting/index.html) (TMM in *U.S. Air Force* magazine  
August and September 1996)

## About the Author

With a M.Sc.C.E. degree, Anne Mette Jonassen Hass has worked in IT since 1980. She has been involved in all aspects of software development: requirements specification, analysis, design, coding, test, quality assurance, and management. Mrs. Hass has worked in various types of businesses such as hospitals, the oil industry, telecommunication, the public sector, and the space industry in Denmark, Norway, England, France, and Italy.

Since 1995 she has worked as a senior consultant in DELTA AXIOM. Mrs. Hass has worked in the fields of software testing, requirements management, configuration management, and maturity assessments.

Mrs. Hass holds the ISEB Foundation certificate in software testing and the ISEB Practitioner certificate in software testing. She has acted as a consultant in the improvement of test procedures and the specification and execution of software test in many companies. She also undertakes third-party testing and validation of software, especially safety-critical software.

Mrs. Hass is the president of the iNTCCM, International Certified Configuration Manager, and she has been heavily involved in the writing of the syllabus for the configuration management certification, foundation level.

Mrs. Hass is in charge of the courses provided by herself and other instructors in DELTA AXIOM. She produces training and marketing material, runs courses, and oversees administrative activities. DELTA AXIOM's courses in ISTQB/ISEB software testing foundation and ISEB software testing practitioner developed by Mrs. Hass are accredited in both English and Danish. Currently Mrs. Hass is working on the courses for the ISTQB advanced certifications. DELTA AXIOM's course in configuration management, also developed by Mrs. Hass, is accredited by the iNTCCM.

Mrs. Hass is certified ISO 15504 lead assessor, and she has performed more than 40 assessments in Denmark, Canada, and Poland for companies of all sizes and in many different branches.

Mrs. Hass is the secretary of the Danish Special Interest Group for Software Test and Test Management and has been running this group successfully since 1997.

Mrs. Hass is a frequent speaker at national and international conferences and has solid experience in teaching at many levels. She has been the Danish country coordinator for the EuroSTAR conference for five years and was on the program committee for the 2003 conference. Mrs. Hass was also on the program committee for EuroQUEST 2007 and will be in the committee for EuroQUEST 2008 as well.

Mrs. Hass is the author of the books *Configuration Management Principles and Practice* (Addison-Wesley, 2003) and *Requirements Development and Management*, DF-17 2002.

In addition, Mrs. Hass has developed the team game "Process Contest," which provides a fun way to learn development concepts and terms.

She is also creator of the posters "... at a Glance—or two" now covering "Software Testing," and "Configuration Management," and soon to cover also "Requirements Handling," "Process Improvement," and "Project Management."

# Index

## A

Acceptance testing, 15–16  
alpha test, 16  
beta test, 16  
contract test, 16  
defined, 15  
goal, 15  
planning, 41  
techniques, 15  
Accessibility, 251–52  
Accuracy testing, 246–47  
Action, incident, 317–18  
Adaptability testing, 272–73  
Agile development models, 7  
Allpairs algorithm, 190  
Alpha tests, 16  
Analogies, 110  
Analyzability testing, 269  
Anomalies  
classification, 324–25  
defined, 311  
*See also* Incidents  
API testing tools, 388  
Attacks, 221–22  
defined, 221  
media-based, 222  
stored data, 222  
user interface, 222  
Attractiveness, 251  
Audits, 300–301

## B

Basic blocks, 199  
BCS working group, 255  
Best guess, 109  
Beta tests, 16  
Big-bang integration, 13  
BOOTSTRAP, 337  
Borders  
closed, 161, 162  
coverage elements, 163  
open, 161  
Bottom-up integration, 13  
Boundary values, 154–56  
defects and, 155  
defined, 154  
identifying, 154  
testing, 155–56  
*See also* Equivalence partitioning and boundary value analysis  
Brainstorming, 134  
British Standard (BS), 332  
Bug tracking applications, 387–88  
Business value, 79–85  
decision improvement, 84–85  
process improvement, 85  
product improvement, 81–84

## C

Call graphs, 232–33  
Capability Maturity Model<sup>®</sup>. *See* CMM<sup>®</sup>; CMMI<sup>®</sup>  
Cause-effect graphs, 169–73  
coverage, 170

- Cause-effect graphs (continued)
  - defined, 169
  - example, 171–72
  - graphing process, 170–71
  - hints, 172–73
  - size problem, 173
  - template, 170–71
  - See also* Specification-based techniques
- Changeability, 270
- Change control process, 327–28
- Change management, 22
- Checklist-based testing, 216–18
  - CRUD, 217
  - defined, 217
  - example, 217–18
- Check sheets, 120–21
- Chunking, 292
- Classification trees, 179–86
  - coverage, 181
  - creation, 180–81
  - example, 182–85, 241
  - hints, 186
  - leaves, 181
  - nodes, 179
  - rules, 180
  - test design template, 181–82
  - uses, 186
  - See also* Specification-based techniques
- Closed borders, 161, 162
- CMM<sup>®</sup>, 338–39
  - defined, 338
  - inadequacy, 339
  - key process areas (KPAs), 338
  - levels, 336
  - TMM and, 346–47
- CMMI<sup>®</sup>, 329, 339–40
  - continuous representation, 339
  - process areas, 339
  - representations, 339
  - staged representation, 339
- Code complexity, 227
- Code metrics calculation, 227–30
- Coding standard compliance, 227
- Coexistence testing, 272
- Coincidental correctness, 215
- Collapsed decision tables, 169
- Communication
  - incidents, 321–22
  - with project management, 406
- team, 405–7
- Communications testing tools, 388
- Comparable measurements, 31
- Comparison tools, 379
- Completion criteria, 44–45
- Component testing, 9–11
  - component test plan, 10
  - execution, 11
  - goal, 10
  - risk exposure and, 130
  - techniques, 10
- Conditions
  - coverage, 203
  - in decision statements, 202
  - defined, 199
- Condition testing, 202–4
  - determination, 205
  - multiple, 204–5
- Confidence measurements, 30
- Confidentiality, creating, 31
- Configuration management, 21–23
  - approach to, 95–96
  - change control, 22
  - defined, 21
  - identification, 21
  - interface with, 44
  - status reporting, 22
  - storage, 22
  - tool support, 372–73
- Confirmation testing, 70, 95
- Contingency planning, 131, 143
- Contract acceptance test, 16
- Control flow analysis, 223–24
- Couplings, 232
- Coverage
  - analysis, 234
  - cause-effect graphs, 170
  - checklist-based testing, 216–17
  - classification trees, 181
  - completion criteria, 45
  - condition, 203
  - decision tables, 167
  - defined, 47, 151
  - domain analysis, 163
  - LCSAJ, 206, 208
  - measurements, 30
  - pairwise testing, 187
  - path, 209
  - statement testing, 200

- state transition testing, 174–75  
structure-based techniques, 198  
use case testing, 193  
CRUD checklists, 217  
CTPs (Critical Testing Processes), 329, 352–55  
    assessment, 354  
    critical processes, 353  
    defined, 39, 352–53  
    improving, 355  
    testing process example, 353–54
- D**
- Dansk Standard (DS), 332  
Database testing tools, 386  
Data flow  
    analysis, 224–27  
    anomalies, 224  
    design test cases, 226  
    phases, 224  
    usage, 225  
Data preparation tools, 373–74  
Debugging tools, 382–83  
Decision/branch testing, 201–2  
Decision improvement, 84–85  
Decision outcome, 199  
Decision tables, 166–69  
    collapsed, 169  
    coverage measure, 167  
    defined, 166  
    example, 168  
    templates, 167  
    uses, 166  
    *See also* Specification-based techniques  
Defect-based techniques, 211–14  
    fault injection and mutation, 213–14  
    taxonomies, 211–13  
    *See also* Test case design techniques  
Defect Detection Percentage (DDP), 321  
Defects  
    boundary values and, 155  
    correction, xxix  
    early detection, 279  
    identifying, 143  
    life cycle, 313–19  
    product risks and, 143  
Delphi technique, 110–11  
Deployment, test tool, 369  
Developers, 136
- Development models, 2–7  
    incremental, 5–7  
    iterative, 5–7  
    sequential, 3–5  
Disposition, incident, 318–19  
Domain analysis, 160–66  
    coverage, 163  
    strategy, 163  
    test design example, 165–66  
    test design template, 164  
    *See also* Specification-based techniques
- Domains  
    borders, 161, 162  
    defined, 160
- Domain-specific standards, 332–33
- Dynamic analysis, 233–35  
    advantages, 233  
    coverage analysis, 234  
    defined, 233  
    memory handling, 233–34  
    memory leaks, 233–34  
    performance analysis, 235  
    pointer handling, 234  
    tools, 381–82  
    *See also* Static analysis
- Dynamic testing, 276
- Dynamic test levels, 8–16  
    acceptance testing, 15–16  
    component testing, 9–11  
    integration testing, 12–14  
    system testing, 14–15
- E**
- Easter eggs, 260  
Effects, 137–40  
    analysis, 139–40  
    defined, 137  
    failure, 138  
    indirect losses, 138  
    scores, 139  
Efficiency testing, 265–68  
    performance, 265–68  
    resource utilization, 268  
Emulation tools, 375–76  
End users, perception, 136  
Equivalence partitioning and boundary value analysis, 152–60  
    boundary value analysis, 154–56

Equivalence partitioning and boundary value  
(continued)  
equivalence partitioning, 153–54, 160  
hints, 160  
test design examples, 157–59  
test design template, 156  
*See also* Specification-based techniques  
Error guessing, 215–16, 215–17  
Estimation  
as predictions, 107  
principles, 106–7  
test, 106–15  
Evolutionary development models, 7  
Execution tools, 376–77  
Exit criteria, evaluating, 71–74  
Experience-based testing techniques, 214–22  
attacks, 221–22  
checklist-based, 216–18  
defined, 215  
error guessing, 215–16  
exploratory testing, 218–21  
*See also* Test case design techniques  
Expert interviews, 134  
Experts, 110  
Exploratory testing, 218–21  
defined, 218  
degrees, 219  
hints, 220–21  
performing, 219–20  
requirements, 218  
weaknesses, 220–21  
*See also* Experience-based testing techniques  
eXtreme Programming model, 7

**F**

Failures, identifying, 68

Fault injection, 213–14

Fault-seeding/fault-injection tools, 379–80

Functional security testing, 248

Functional testing, 245–48

accuracy, 246–47

interoperability, 247–48

security, 248

suitability, 245–46

tools, 385

Function points, 112–13

**G**

Gelperin-Hetzl historical model, 343  
Graphics, 117

**H**

Herzberg's factors, 403–4

**I**

IEEE 1044, 313, 314, 319  
classification scheme, 319  
Standard Classification for Software  
Anomalies, 324–25  
IEEE standards, 332  
Incident management, 311–22  
approach to, 95  
metrics and, 319–21  
Incidents  
action, 317–18  
causes, 312  
classification scheme, 319  
communicating, 321–22  
defined, 311  
detection, 311–13  
disposition, 318–19  
distribution, 321  
fields, 319  
investigation, 315–17  
life cycle, 313–19  
measurements, 30  
recognition, 314–15  
report information, 320  
reporting and tracking, 312–13

Independence, in testing, 399–401

Industry-specific standards, 330

Informal reviews, 284–85

Inspection-based process improvement,  
299–300

Inspection meeting, 296–98

logging form, 298

moderator, 297

purpose, 296–97

Inspections

activities, 290

entry/exit criteria, 291–92

Fagan, 289

follow-up, 299

leader, 291

- overview, 293–94  
planning, 292–300  
preparation, 294–95  
purposes, 290  
rework after, 299  
roles, 290, 293
- Inspector roles, 295–96  
Installability testing, 271–72  
Integration testing, 12–14  
    big-bang, 13  
    bottom-line, 13  
    execution, 13  
    goals, 11  
    summary report, 12  
    techniques, 13  
    top-down, 12
- Intercomponent testing, 210–11  
    combining with other techniques, 211  
    defined, 210  
    use, 210  
    *See also* Structure-based techniques
- International standards, 331–32
- Interoperability testing, 247–48
- Interpersonal skills, 392–94
- Invalid syntax, testing, 196
- ISO 9126 standard, 244, 255, 261
- ISO 15504, 340–41  
    defined, 340  
    development life cycle focus, 341
- ISO standards, 331
- ISTQB advanced certification, xix, xxi
- ISTQB Certified Tester, xix, xx, xxi
- ISTQB Glossary of Terms, xxv
- Iterative/incremental models, 5–7  
    assumptions, 5  
    characteristics, 6  
    defined, 5  
    evolutionary development models, 7
- RAD, 6–7
- Spiral Model, 7
- J**
- Java testing tools, 385–86
- K**
- Key process areas (KPAs), 338
- Keyword-driven automation tools, 377–79  
    advantages, 378–79
- defined, 377  
disadvantages, 379  
keywords, 378
- L**
- LCSAJ (loop testing), 206–8  
    coverage, 206, 208  
    defined, 206  
    example, 207–8  
    test case design, 207
- Learnability, 250
- Level test plan, 97  
    defined, 86  
    size, 97  
    *See also* Test plans
- Lines of code (LOC), 227
- Link and HTML testing tools, 386
- Load testing, 268
- Logging, test execution, 68–69
- Logical bombs, 260
- Loop testing, 206–8
- M**
- Madsen's motivation theory, 404–5
- Maintainability testing, 268–69  
    analyzability, 269  
    changeability, 270  
    stability, 270  
    as static testing, 269  
    testability, 270–71
- Management reviews, 288–89  
    advantages/disadvantages, 289  
    objective, 288  
    roles, 288–89  
    types of, 288
- Maslow's hierarchy of needs, 402–3
- Master test plan, 96–97  
    defined, 86, 96  
    stakeholders, 96–97  
    *See also* Test plans
- Maturity testing, 262–63
- McCabe's Cyclomatic Complexity, 228–30
- Measurement presentation, 116–24  
    check sheets, 120–21  
    graphics and, 117  
    pie charts, 120

- Measurement presentation (continued)  
 risk-based reporting, 121–22  
 S-curves, 118–19  
 statistical reporting, 122–24
- Measurements  
 analysis and presentation of, 31  
 comparable, 31  
 confidence, 30  
 coverage, 30  
 defined, 28  
 direct, 29  
 economical, 31  
 getting, 115  
 incidents, 30  
 planning, 31  
 precise, 31  
 progress, 29–30  
 repeatable, 31
- Media-based attacks, 222
- Metrics  
 agreed, 31  
 defined, 28  
 incident management, 319–21  
 reporting, 73–74  
 subjective, 29  
 test closure, 76  
 test implementation and execution, 71  
 test planning and control, 50  
 test progress, 73–74  
 test-related, 29–30
- Model requirements, 59
- Motivation, 401–5  
 Herzberg's factors, 403–4  
 Madsen's motivation theory, 404–5  
 Maslow's hierarchy of needs, 402–3  
 testers', 405
- Mutation testing, 214
- N**  
 National standards, 332
- O**  
 Online link and HTML testing services, 386–87  
 Open borders, 161  
 Operability, 250–51  
 Optimized expressions, 205  
 Oracles, 374–75
- advantages/disadvantages, 375  
 automated, 374  
 defined, 374
- Organizational anchorage, 398–99
- Orthogonal arrays, 187–90  
 as balanced, 187  
 creating, 188  
 defined, 187  
 example, 189–90  
 size/contents description, 187–88
- P**
- Pair testing, 221
- Pairwise testing, 186–91  
 allpairs algorithm, 190  
 coverage, 187  
 defined, 187  
 higher-order combinations, 190  
 hints, 191  
 orthogonal arrays, 187–90  
*See also* Specification-based techniques
- Path testing, 209–10  
 coverage, 209  
 error guessing, 210
- People skills, 389–407  
 communication, 405–7  
 individual, 389–94  
 motivation, 401–5  
 test team dynamics, 394–97
- Percentage distribution, 113–14
- Performance analysis, 235
- Performance testing, 265–68  
 defined, 265  
 drawbacks, 266  
 load, 266  
 scalability, 268  
 stress, 267  
 tools, 385
- Pie charts, 120
- Pointer handling, 234
- Portability testing, 272–73  
 adaptability, 272–73  
 coexistence, 272  
 defined, 271  
 installability, 271–72
- Precise measurements, 31
- Probability analysis, 140–41
- Processes, 33–76

- concept, 34  
dependence, 35  
generic test, 35–38  
improvement models, 39  
input to, 35  
monitoring, 34–35
- Process improvement, 333–56  
  approaches, 96, 334  
  basis, 333  
  CTP, 355  
  defined, 329  
  inspection-based, 299–300  
  organization requirements, 335  
  principles, 334–37  
  results, 335–37  
  testing improvement models, 341–56  
  TPI, 351–52  
  urge for, 334  
  value of, 85
- Process maturity models, 337–41  
  CMM<sup>®</sup>, 338–39  
  CMMI<sup>®</sup>, 339–40  
  defined, 337  
  ISO 15504, 340–41
- Process risks, 127
- Product improvement, 81–84
- Product paradigms, 23–28  
  safety-critical systems, 25–28  
  systems of systems, 24–25
- Product risks, 129–30  
  defects, 143  
  examples, 129  
  origination, 129  
  project risks and, 130  
  *See also* Risks
- Progress monitoring, 115–25
- Project management, 20–21
- Project managers, 136
- Project risks, 128
- Q**
- Quality assurance, 17–20  
  defined, xxvi  
  reports, 20  
  standards, 330  
  of test specification, 64–65  
  validation, 18
- verification, 18–19
- Quality attributes, 243–73  
  accuracy, 246–47  
  adaptability, 272–73  
  analyzability, 269  
  changeability, 270  
  coexistence, 272  
  efficiency, 265–68  
  functional testing, 245–48  
  installability, 271–72  
  interoperability, 247–48  
  maintainability, 268–71  
  maturity, 262–63  
  performance, 265–68  
  portability, 271–73  
  random input, 257–58  
  recoverability, 264–65  
  reliability, 261–65  
  replaceability, 273  
  resource utilization, 268  
  robustness, 263–64  
  security, 248  
  stability, 270  
  suitability, 245–46  
  for technical test analysts, 254–73  
  testability, 270–71  
  for test analysts, 244–54  
  testing, 243–73  
  usability, 249–54
- Questionnaires, 254
- R**
- RAD model, 6–7
- Random input generation, 257–58
- Reading guidelines, this book, xx
- Recoverability testing, 264–65
- Regression testing, 70–71  
  amount, 71  
  approach to, 95  
  defined, 70  
  example, 70–71
- Reliability testing, 261–65  
  maturity, 262–63  
  measurement evaluation, 261  
  recoverability, 264–65  
  robustness, 263–64
- Repeatable measurements, 31
- Replaceability testing, 273

- Reporting  
 activities, 72–73  
 completion, input, 72  
 documentation, 72  
 incident, 312–13  
 metrics, 73–74  
 risk-based, 121–22  
 statistical, 122–24  
 status, 22  
*See also* Test reports
- Requirements, 57–59  
 levels, 57–58  
 model, 59  
 statement, 59  
 styles, 58–59  
 table, 59  
 task, 59  
 types, 58
- Requirements management testing tools, 388
- Resource utilization testing, 268
- Retrospective meeting, 75
- Return on investment (ROI), 335
- Reviewing. *See* Static testing
- Reviews  
 informal, 284–85  
 management, 288–89  
 technical, 286–87
- Risk analysis, 135–42  
 effect, 137–40  
 level, 141–42  
 perceptions and, 135–36  
 probability, 140–41  
 repetition, 142  
 results, 130, 142  
 scales for, 136–37  
 template, 135
- Risk-based reporting, 121–22
- Risk-based testing, 125–30
- Risk level, 141–42  
 calculation, 141  
 defined, 126  
 distribution, 142
- Risk management, 131–34  
 activities, 313  
 brainstorming, 134  
 contingency planning, 131  
 defined, 131  
 expert interviews, 134  
 identification, 132–33
- independent assessments, 134  
 lessons learned and checklists, 133  
 testing and, 130  
 workshops, 133
- Risk mitigation, 142–47  
 process, 143  
 by testing, 144–45  
 timing, 145–47
- Risks, 125–47  
 checklists, 133  
 defined, 126  
 identifying, 132–33  
 independent assessments, 134  
 perception, 135–36  
 process, 127  
 product, 129–30  
 project, 128  
 template, 135  
 types of, 127–29  
 workshops, 133
- Robustness testing, 263–64
- Rollout, test tool, 369
- S**
- Safety-critical systems, 25–28  
 software integrity levels (SILs), 27  
 standards, 26–27  
 values, 26
- Sampling, 292
- Scalability testing, 268
- Schedules  
 producing, 49  
 test plan, 102, 103–4
- S-curves, 118–19  
 illustrated, 119  
 phases, 118  
 uses, 118  
*See also* Measurement presentation
- Security testing  
 functional, 248  
 technical, 258–61
- Sequential models, 3–5  
 assumptions, 3  
 characteristics, 3  
 V-model, 4  
 waterfall, 3–4  
 W-model, 4–5
- Simulation tools, 375–76

- Skills  
individual, 389–94  
interpersonal, 392–94  
people, 389–407  
test roles and, 391–94
- Software cycle, testing in, 1–23
- Software integrity levels (SILs), 27
- Software testing. *See* Testing
- Source testing tools, 385
- Specification-based techniques, 152–97  
cause-effect graphs, 169–73  
classification trees, 179–86  
decision tables, 166–69  
defined, 152  
domain analysis, 160–66  
equivalence partitioning and boundary value analysis, 152–60  
focus, 152  
pairwise testing, 186–91  
state transition testing, 173–79  
syntax testing, 193–97  
types of, 152  
use case testing, 191–93  
uses, 152  
*See also* Test case design techniques
- SPICE, 329
- Spiral Model, 7
- Stability testing, 270
- Staffing needs, 102
- Standard Classification for Software Anomalies (IEEE 1044–1993), 324–25
- Standards, 330–33  
defined, 329  
domain-specific, 332–33  
general, 330–31  
IEEE, 332  
industry-specific, 330  
international, 331–32  
ISO, 331  
national, 332  
quality assurance, 330  
sources, 330  
testing, 330
- State machine, 173–74
- Statements  
decision, 202–3  
defined, 199  
execution, 199–200  
requirements, 59
- Statement testing, 199–201  
coverage, 200  
defined, 199  
example, 200–201  
*See also* Structure-based techniques
- State transition testing, 173–79  
coverage, 174–75  
defined, 173  
example, 177–78  
hints, 179  
templates, 175–76  
test conditions, 175  
*See also* Specification-based techniques
- Static analysis, 222–33  
of architecture, 230–33  
audit, 300–301  
calculation of code metrics, 227–30  
call graphs, 232–33  
of code, 223–30  
compliance to coding standard, 227  
control flow, 223–24  
data flow, 224–27  
defined, 222  
inspection, 289–92  
inspection planning, 292–300  
maintainability, 269  
stakeholders, 230  
tools, 380–81  
usability, 253  
of Web sites, 230–32  
*See also* Dynamic analysis
- Static testing, 275–306  
author, 282  
champion, 304  
change agents, 304  
checking, 280–81  
of code, 283–84  
cost/benefit, 278–79  
decision maker, 282  
defined, 275, 276  
general principles, 275–84  
generic process, 279–81  
history, 275–76  
implementation process, 303  
implementation roles, 303–4  
informal review, 284–85  
introduction of, 303–6  
leader, 282  
in life cycle, 301–2

- Static testing (continued)
- management review, 288–89
  - objective, 276
  - objects, 277–78
  - outcome, 281
  - piloting, 305
  - planning, 301
  - processes, 304–5
  - psychological aspects, 306
  - reader, 282
  - recorder, 282
  - of requirements specifications, 302
  - roles in, 281–82
  - rollout, 305–6
  - target group, 304
  - technical review, 286–87
  - test processes applied to, 279–80
  - type mixer, 283
  - types, 284–301
  - type selection, 282–84
  - walk-through, 285–86
- Statistical reporting, 122–24
- defined, 122
  - norm, 122–23
- Status reporting, 22
- STEP (Systematic Test and Evaluation Process), 329, 355–56
- defined, 355
  - maintenance testing, 356
  - steps, 355
  - use of, 356
- Storage, 22
- Stored data attacks, 222
- Stress testing, 267
- Structure-based techniques, 197–211
- condition determination testing, 105
  - decision/branch testing, 201–2
  - defined, 197
  - intercomponent testing, 210–11
  - LCSAJ (loop testing), 206–8
  - multiple condition testing, 204–5
  - path testing, 209–10
  - statement testing, 199–201
  - test coverage, 198
  - types of, 198
  - white-box concepts, 198–99
- See also* Test case design techniques
- Structured testing, 67
- Suitability testing, 245–46
- SUMI (Software Usability Measurement Inventory), 254
- Supporting processes, 16–23
- Surveys, 254
- Syntax testing, 193–97
- coverage measure and, 194
  - defined, 194
  - example, 195–97
  - hints, 197
  - mutations, 194
  - rules, 194
  - templates, 195
- See also* Specification-based techniques
- Systems of systems, 24–25
- complexity, 24
  - defined, 24
  - examples, 24, 25
  - weakest link, 25
- System testing, 14–15
- execution, 14–15
  - goal, 14
  - report, 15
  - specification, 14
  - techniques, 14
  - tools, 15
- T**
- Table requirements, 59
- Task requirements, 59
- Taxonomies, defect, 211–13
- Technical reviews, 286–87
- advantages/disadvantages, 287
  - defined, 286
  - roles, 287
- Technical testing, 254–73
- efficiency, 265–68
  - general, 256–57
  - Maintainability, 268–71
  - portability, 271–73
  - random input technique, 257–58
  - reliability, 261–65
  - security, 258–61
- Technical writing
- defined, 23
  - testing interface, 16–17
- Templates
- cause-effect graphs, 170–71
  - classification tree method, 181–82

- decision table, 167
- domain analysis, 164
- equivalence partitioning and boundary value analysis, 156
- risk, 135
- state transition testing, 175–76
- syntax testing, 195
- Testability testing, 270–71
- Test analysis and design, 50–61
  - activities, 51–57
  - design definition, 52–53
  - documentation, 51
  - inputs, 50, 51
  - metrics, 61
  - output, 51
  - requirements, 57–59
  - test cases, 54–57
  - test conditions, 53–54
  - traceability, 60–61
- Test automation
  - approach to, 94
  - complexity, 361
- Test basis, 42–43
- Test case design techniques, 43, 54–55, 93, 151–237
  - choosing, 235–37
  - defect-based, 211–14
  - dynamic analysis, 233–35
  - experience-based, 214–22
  - pitfalls, 151
  - selection advice, 236–37
  - specification-based, 152–97
  - static analysis, 222–33
  - structure-based, 197–211
  - subsumes ordering of, 236
- Test cases
  - creation of, 54–57
  - data flow technique, 226
  - for decision testing, 201
  - expected results, 56
  - high-level, 54, 55
  - low-level, 55–56
  - in test procedures, 63
- Test closure, 74–76
  - activities, 74, 75
  - documentation, 74
  - inputs, 74
  - metrics, 76
- Test conditions
- coverage items and, 53
- extraction, 53
- identification of, 53–54
- Test designs
  - contents, 52
  - defining, 52–53
  - specification identifier, 52
  - tools, 373
- Test environment
  - as execution prerequisite, 65
  - problems with, 66
  - specification, 65–66
- Testers
  - ethics, xxii
  - job, xxix
  - perception, 136
- Test estimation, 106–15
  - analogies and experts, 110
  - best guess, 109
  - Delphi technique, 110–11
  - function points, 112–13
  - percentage distribution, 113–14
  - principles, 107–8
  - process, 108–9
  - techniques, 109–14
  - test points, 113
  - three-point, 111–12
- Test group, 52
- Test implementation and execution, 61–71
  - activities, 61–71
  - confirmation testing, 70
  - documentation, 62
  - entry criteria, 66–71
  - environment specification, 65–66
  - failure identification, 68
  - inputs, 61, 62
  - logging, 68–69
  - metrics, 71
  - output, 62
  - procedure organization, 62–65
  - regression testing, 70–71
- Testing
  - acceptance, 15–16
  - accuracy, 246–47
  - adaptability, 272–73
  - analyzability, 269
  - basics, xxiii–xxix, 1–31
  - business value, 79–85
  - changeability, 270

- Testing (continued)  
  checklist-based, 216–18  
  coexistence, 272  
  component, 9–11  
  condition, 202–4  
  condition determination, 205  
  confirmation, 70, 95  
  decision/branch, 201–2  
  defined, xxv–xxvi  
  dynamic, 276  
  efficiency, 265–68  
  exploratory, 218–21  
  fitting into organization, 398–401  
  functional, 245–48  
  functional security, 248  
  independence in, 399–401  
  installability, 271–72  
  integration, 11, 12–14  
  intercomponent, 210–11  
  interoperability, 247–48  
  load, 268  
  loop, 206–8  
  maintainability, 268–71  
  maturity, 262–63  
  as multidimensional, xxiv–xxv  
  multiple condition, 204–5  
  mutation, 214  
  necessity of, xxvi–xxix, 2  
  pair, 221  
  pairwise, 186–91  
  path, 209–10  
  performance, 265–68  
  processes, 33–76  
  on products/work products, xxv  
  purpose of, 80  
  recoverability, 264–65  
  regression, 70–71, 95  
  reliability, 261–65  
  replaceability, 273  
  resource utilization, 268  
  risk and, 125–47  
  risk management and, 130  
  robustness, 263–64  
  scalability, 268  
  security, 248, 258–61  
  in software cycle, 1–23  
  stability, 270  
  standards, 330  
  statement, 199–201  
    state transition, 173–79  
    static, 275–306  
    stress, 267  
    structured, 67  
    suitability, 245–46  
    syntax, 193–97  
    system, 14–15  
    technical, 254–73  
    technical security, 258–61  
    technical writing and, 16–17  
    terms and definitions, xxiii  
    testability, 270–71  
    usability, 249–54  
    use case, 191–93
- Testing improvement models, 341–56  
  STEP, 355–56  
  TMM, 329, 342–47  
  TPI, 347–52
- Testing tools, 361–83  
  acquisition, 362–67  
  for all testers, 371–73  
  API, 388  
  bug tracking applications, 387–88  
  business case preparation, 363–64  
  buying, 365–66  
  categories, 370–83  
  classification, 370–71  
  communications, 388  
  comparison, 379  
  for configuration management, 372–73  
  database, 386  
  data preparation, 373–74  
  debugging, 382–83  
  deployment, 369  
  design, 373  
  do-it-yourself, 365–66  
  dynamic analysis, 381–82  
  emulation, 375–76  
  environmental requirements, 365  
  evaluation, 366–67  
  execution, 376–78  
  fault-seeding/fault-injection, 379–80  
  functional, 385  
  functionality, 364  
  implementation process, 368  
  Java testing, 385–86  
  keyword-driven automation, 377–79  
  link and HTML, 386  
  list of, 385–88

- nonfunctional requirements, 364
- online link and HTML, 386–87
- open-source, 365–66
- oracles, 374–75
- performance of competitive trials, 367
- performance testing, 382, 385
- piloting, 368–69
- for programmers, 382–83
- project requirements, 365
- purpose, 361
- requirements, 364–65
- requirements management, 388
- rollout, 369
- selection team, 363
- shortlist preparation, 366
- simulation, 375–76
- source, 385
- static analysis, 380–81
- strategy, 363
- for technical test analysts, 380–82
- to test analysts and technical analysts, 373–80
- test management, 371
- usability, 364
- use decision, 362
- Web, 380
- Web-based bug tracking, 387
- Web performance, 387
- Web security, 387
- Test invalid conditions, 176
- Test management, 79–147
  - business value and, 79–85
  - control, 124–25
  - estimation, 106–15
  - progress monitoring and control, 115–25
  - risk and, 125–47
  - tools, 371
- Test management documentation, 85–106
  - elements, 85
  - higher management, 86–106
  - level test plan, 85, 97
  - master test plan, 86, 96–97
  - overview, 85–86
  - project level, 96–106
  - test policy, 86, 86–88
  - test strategy, 86, 88–96
- Test objects, xxvii, 42–43
- Test planning and control, 39–50
  - activities, 40, 41–50
- approach definition, 43–44
- completion criteria definition, 44–45
- documentation, 41
- environment outline, 65
- inputs, 40–41
- metrics, 50
- output, 40
- purpose, 39–40
- scoping test effort, 47–49
- starting early, 103–4
- test basis, 42–43
- test object definition, 42–43
- work products definition, 45–47
- Test plans
  - approach, 100
  - defined, 96
  - deliverables, 101
  - environmental needs, 101
  - features not to be tested, 99–100
  - features to be tested, 99
  - introduction, 98–99
  - item pass/fail criteria, 100
  - level, 97
  - master, 86, 96–97
  - responsibilities, 102
  - risks and contingencies, 103
  - schedule, 102
  - staffing/training needs, 102
  - suspension/resumption, 100–101
  - template, 97–103
  - testing tasks, 101
  - test items, 99
- Test points, 113
- Test policy, 86–88
  - approach to test process improvement, 88
  - defined, 86
  - definition of testing, 87
  - evaluation of testing, 87–88
  - quality targets, 88
  - testing process, 87
- Test procedures
  - defined, 62
  - documentation, 63
  - example template, 64
  - organizing, 62–65
- See also* Test cases
- Test process, 35–38
  - activities, 36
  - dependencies, 37

- Test process (continued)
- inputs, 36
  - iterations, 38
  - outputs, 36
  - purpose, 35
  - static, 36
- See also* Processes
- Test progress, 71–73
- activities, 72–73
  - documentation, 72
  - inputs, 72
  - metrics, 73–74
  - output, 72
- Test progress monitoring and control, 115–25
- data collection, 116
  - measurement presentation, 116–24
- Test reports, 104–6
- comprehensiveness assessment, 105
  - evaluation, 106
  - identifier, 104–5
  - summary, 105
  - summary of activities, 106
  - summary of results, 205–6
  - in test progress communication, 116
  - test team, 395–96
  - variances, 105
- See also* Reporting
- Test roles
- defining, 48–49
  - skills and, 391–94
  - test team, 49
  - types of, 49
- Test specification
- environment, 65–66
  - quality assurance of, 64–65
  - structure, 43, 46
- Test strategy, 88–96
- approach to configuration management, 95–96
  - approach to confirmation/regression testing, 95
  - approach to incident management, 95
  - approach to test automation, 94
  - approach to test process improvement, 96
  - defined, 86, 88
  - degree of independence, 92–93
  - environment, 94
  - extent of reuse, 93–94
  - identifier, 90
- introduction, 90–91
- level entry criteria, 92
  - level exit criteria, 92
  - measure to be captured, 94–95
  - risks, 91
  - standards, 91
  - test case design techniques, 93
  - test levels, 91–92
- Test teams
- aspects, 397
  - communication, 405–7
  - dynamics, 394–97
  - forming, 397
  - roles, 49, 395–96
- Test ware
- approach to, 95–96
  - archiving, 76
  - delivering, 75
- Three-point estimation, 111–12
- TMM (Testing Maturity Model), 329, 342–47
- assessment model, 346
  - CMM® and, 346–47
  - defined, 342
  - development, 343
  - mature testing process attributes, 344
  - maturity levels, 344–46
  - staging, 345
  - stakeholders using, 342
  - structure, 342
- Tools for technical test analysts, 380–82
- dynamic analysis tools, 381–82
  - performance testing tools, 382
  - static analysis tools, 380–81
- See also* Testing tools
- Tools for test analysts/technical test analysts, 373–80
- comparison tools, 379
  - emulation tools, 375–76
  - fault-seeding/fault-injection tools, 379–80
  - keyword-driven automation tools, 377–79
  - simulation tools, 375–76
  - test data preparation tools, 373–74
  - test design tools, 373
  - test execution tools, 376–77
  - test oracles, 374–75
  - Web tools, 380
- See also* Testing tools
- Top-down integration, 12
- Total quality model (TQM), 347

TPI (Test Process Improvement Model),  
329, 347–52  
assessment, 350–51  
checkpoints, 352  
cornerstones, 348  
defined, 39, 347  
key areas, 348  
levels, 349–50, 358–59  
process improvement, 351–52  
structure, 347–48  
Traceability, 60–61  
Training needs, 102

## U

Understandability, 250  
Usability  
accessibility, 251–52  
assessment, 252  
requirements, establishing, 252–53  
static tests, 253  
subattributes, 250–51  
testing, 253–54  
test tool, 364  
users concerned with, 249  
verification and validation, 253  
Use cases  
coverage, 193

defined, 191  
description, 193  
example, 192  
structured textual form, 191  
testing, 191–93  
User interface attacks, 222

## V

Validation, 18  
Verification, 18–19  
V-model, 4  
defined, 4  
dynamic test levels, 8

## W

Walk-throughs, 285–86  
WAMMI, 254  
Waterfall model, 3–4, 301–2  
Web-based bug tracking, 387  
Web sites, static analysis of, 230–32  
Web tools, 380, 387  
White-box concepts, 198–99  
W-model, 4–5  
Work breakdown structure, 48  
Work products, 45–47  
Workshops, 133