PRACTICAL TEST DESIGN

Selection of traditional and automated test design techniques

István Forgács and Attila Kovács
PRACTICAL TEST DESIGN
BCS, THE CHARTERED INSTITUTE FOR IT

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

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

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

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

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

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

I. Forgács and A. Kovács
## CONTENTS

List of figures and tables ix  
Authors xiii  
Foreword xiv  
Acknowledgements xvi  
Abbreviations xvii  
Glossary xviii  
Letter to the reader xxi  
Preface − Testing is complex xxii

1. **INTRODUCTION**  
The importance of software testing 1  
What is test design exactly? 2  
Why should tests be designed? 17  
When do we design tests? 18  
Important test design-related considerations 19  
Summary 21

2. **EXAMPLE SPECIFICATION: TICKET VENDING MACHINE**  
Summary 26

PART I  
**NECESSARY STEPS BEFORE TEST DESIGN** 27

3. **RISK MANAGEMENT AND TEST OPTIMISATION** 29  
What is risk? 29  
Scoring the risks 30  
Risk management 32  
Risks, costs and quality 34  
Risk and optimisation 38  
Example: risk analysis of TVM 39  
Key takeaways 43

4. **DEFECT PREVENTION** 44  
Defect prevention methods 46  
From requirements to specification (‘Two lifts in a 10-storey building’ example) 47  
Refinement of the ticket vending machine specification 51  
Key takeaways 54
## PART II TRADITIONAL TEST DESIGN

### 5. DOMAIN-BASED TESTING
- Equivalence partitioning
- Boundary value analysis
- Domain analysis
- Challenges in domain-based testing
- TVM example
- Method evaluation
- Theoretical background
- Key takeaways
- Exercises

### 6. STATE TRANSITION TESTING
- Stateful and stateless systems
- States, transitions, conditions
- Example: RoboDog
- Validate your state transition graph
- Test selection criteria for state transition testing
- All-transition-state criterion
- All-transition–transition criterion
- Example: Collatz conjecture
- When multiple techniques are used together
- State transition testing in TVM example
- Method evaluation
- Theoretical background
- Key takeaways
- Exercises

### 7. BUSINESS RULE-BASED TESTING
- Decision table testing
- Cause–effect graphs
- Method evaluation
- Theoretical background
- Key takeaways
- Exercises

### 8. SCENARIO-BASED TESTING
- Use cases
- User stories
- Methods evaluation
- Theoretical background
- Key takeaways
- Exercises

### 9. COMBINATIVE AND COMBINATORIAL TESTING
- Combinative techniques
- Combinatorial techniques
- Comparison of the techniques
<p>| CONTENTS |
|-----------------|--------|
| Key takeaways     | 268    |
| Exercise          | 268    |
| 14. TESTING THE TVM – SUMMARY AND METHOD COMPARISON | 269 |
| Testing TVM by STT, EP and BVA | 269 |
| Testing TVM by use case testing, EP and BVA | 282 |
| Combinative testing | 282 |
| Testing TVM by other methods | 283 |
| Comparison        | 283    |
| 15. CONCLUSIONS AND RECOMMENDATIONS | 285 |
| Appendix A – TVM models | 288 |
| Appendix B – Test code for demonstrating GraphWalker | 291 |
| Appendix C – POM.XML for GraphWalker | 294 |
| Appendix D – Solutions to the exercises | 297 |
| References        | 321    |
| Index             | 326    |</p>
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Test design in the test software development life cycle</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Selecting five standard tickets in the TVM interface</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Payment process with different possibilities</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Payment snapshot after clicking the €5 in the previous screen</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Payment can be finished</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Successful transaction</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>‘Reduced’ mode</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Payment in ‘Reduced’ mode</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Risk diagram and risk matrix</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Optimum cost of testing. The optimal cost of testing is the sum of the testing cost and the late correcting cost</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Total cost optimum of a high-risk function. The total cost optimum of a high-risk function is above the total cost optimum of a low-risk function</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Defect prevention cycle</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Basic usage of the lifts in the ‘Two lifts in a 10-storey building’ example</td>
<td>49</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Equivalence partitioning and test data selection</td>
<td>59</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Equivalence partitioning with ON/OFF/OUT points for open boundary</td>
<td>65</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Equivalence partitioning with ON/OFF/IN points for closed boundary</td>
<td>66</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Equivalence partitioning with ON/OFF/OUT points for closed boundary from both sides</td>
<td>67</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Equivalence partitioning with ON/OFF/IN points for open boundary from both sides</td>
<td>68</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Valid and invalid partitions with ON-boundary points</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Linearly ordered partitions</td>
<td>73</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>State transition diagram</td>
<td>91</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>State diagram for the ‘RoboDog’ example</td>
<td>95</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>State diagram for the extended ‘RoboDog’ example</td>
<td>96</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Programmed behaviour of the ‘RoboDog’</td>
<td>103</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Collatz machine</td>
<td>106</td>
</tr>
<tr>
<td>Figure 6.6a</td>
<td>State transition graph for ‘Two lifts in a 10-storey building’ – Lift1</td>
<td>111</td>
</tr>
<tr>
<td>Figure 6.6b</td>
<td>State transition graph for ‘Two lifts in a 10-storey building’ – Lift2</td>
<td>112</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Ticket selection transition graph for the TVM</td>
<td>121</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>Dependence diagram for ‘RedShoe’ order handling</td>
<td>140</td>
</tr>
</tbody>
</table>
Figure 7.2  Some standard operators used in cause–effect graphs  147
Figure 7.3  Representations used in cause–effect graphs  147
Figure 7.4  'Exclusive' constraint for cause–effect graphs  148
Figure 7.5  'Inclusive' constraint for cause–effect graphs  148
Figure 7.6  'Require' constraint for cause–effect graphs  149
Figure 7.7  The 'Only' constraint for cause–effect graphs  149
Figure 7.8  'Mask' constraint for cause–effect graphs  149
Figure 7.9  Cause–effect graph for 'RedShoe' special  151
Figure 8.1  Use case graph for 'Enter/Leave Building Zone'  161
Figure 9.1  Fault detection at interaction strength according to Kuhn et al. (2008)  180
Figure 9.2  Classification tree for the ticket selection process  199
Figure 11.1  Iterative process of MBT  218
Figure 11.2  Multi-layered state transition graph  227
Figure 11.3  State transition graph for Sub  227
Figure 12.1  The model of price reduction  239
Figure 12.2  State transition graph for the TVM  247
Figure 13.1  Screen when executing Maven to start GraphWalker  258
Figure 13.2  Directory structure of GraphWalker  258
Figure 13.3  SmallTest.java in the directory structure  259
Figure 13.4  State transition graph of TVM made by yEd  260
Figure 13.5  Test automation pyramid  262
Figure 13.6  Extended state transition graph for TVM  264
Figure 13.7  TVM source Java files  266
Figure 14.1  Statechart for the ticket selection process  271
Figure 14.2  Statechart for the TVM  271
Figure D.1  STT for solving Exercise 6.2  303
Figure D.2  Parameter selection demonstration for Exercise 9.2  314

Table 1.1  Test bases, test design categories and test design techniques  6
Table 1.2  Coverage domains with adequacy criteria  11
Table 3.1  Test design techniques applied for a given aggregated risk class  33
Table 3.2  Risk table for TVM  41
Table 5.1  Equivalence partitioning for the authorisation example  62
Table 5.2  Test design against predicate faults. BVA for predicate Age > 42  64
Table 5.3  Test design against predicate faults. BVA for predicate Age >= 43  66
Table 5.4  Test design against predicate faults. BVA for predicate Age == 43  67
Table 5.5  Test design against data faults. BVA for badVariable >= 43  69
Table 5.6  Test design against data faults. BVA for badVariable == 43  69
Table 5.7  Test design against data faults. BVA for bad constant  70
Table 5.8  Test design against data faults. BVA for other bad predicate  71
Table 5.9  Test design for the 'Authorisation' example  75
Table 5.10  Test cases for the 'Authorisation' example  77
Table 5.11  Test design for the 'Taxi driver' example  81
Table 5.12  Acceptable banknotes for TVM  83
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.13</td>
<td>Acceptable banknotes for TVM – improved</td>
<td>84</td>
</tr>
<tr>
<td>Table 5.14</td>
<td>Test cases for TVM</td>
<td>84</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>State–event notation for a state transition table</td>
<td>92</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>State–state notation for a state transition table</td>
<td>92</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>State–event notation for the ‘RoboDog’ example with undefined transitions</td>
<td>95</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>State–event notation for the ‘RoboDog’ example</td>
<td>95</td>
</tr>
<tr>
<td>Table 6.5</td>
<td>State–event notation for the extended ‘RoboDog’ example</td>
<td>96</td>
</tr>
<tr>
<td>Table 6.6</td>
<td>Short test for the extended ‘RoboDog’ example</td>
<td>104</td>
</tr>
<tr>
<td>Table 6.7</td>
<td>State transition test for ‘Two lifts in a 10-storey building’, ST-Path1</td>
<td>113</td>
</tr>
<tr>
<td>Table 6.8</td>
<td>State transition test for ‘Two lifts in a 10-storey building’, ST-Path2</td>
<td>113</td>
</tr>
<tr>
<td>Table 6.9</td>
<td>BVA test input for ‘Two lifts in a 10-storey building’, ST-Path1</td>
<td>115</td>
</tr>
<tr>
<td>Table 6.10</td>
<td>BVA test input for ‘Two lifts in a 10-storey building’, ST-Path2</td>
<td>116</td>
</tr>
<tr>
<td>Table 6.11</td>
<td>State transition tests for the TVM-Test-1</td>
<td>122</td>
</tr>
<tr>
<td>Table 6.11a</td>
<td>State transition tests for the TVM-Test-2</td>
<td>123</td>
</tr>
<tr>
<td>Table 6.11b</td>
<td>State transition tests for the TVM-Test-3</td>
<td>124</td>
</tr>
<tr>
<td>Table 6.11c</td>
<td>State transition tests for the TVM-Test-4</td>
<td>125</td>
</tr>
<tr>
<td>Table 6.11d</td>
<td>State transition tests for the TVM-Test-5</td>
<td>125</td>
</tr>
<tr>
<td>Table 6.11e</td>
<td>State transition tests for the TVM-Test-6</td>
<td>125</td>
</tr>
<tr>
<td>Table 6.11f</td>
<td>State transition tests for the TVM-Test-6</td>
<td>125</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Limited-entry decision table – first step</td>
<td>130</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Limited-entry decision table – second step</td>
<td>132</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>Limited-entry decision table – third step</td>
<td>133</td>
</tr>
<tr>
<td>Table 7.4</td>
<td>Extended-entry decision table – first step</td>
<td>134</td>
</tr>
<tr>
<td>Table 7.5</td>
<td>Extended-entry decision table – final version</td>
<td>134</td>
</tr>
<tr>
<td>Table 7.6</td>
<td>Extended-entry decision table for the TVM example</td>
<td>137</td>
</tr>
<tr>
<td>Table 7.7a</td>
<td>Top level (Level 1) decision table for ‘RedShoe’</td>
<td>140</td>
</tr>
<tr>
<td>Table 7.7b</td>
<td>Level 2 decision table for ‘RedShoe’</td>
<td>141</td>
</tr>
<tr>
<td>Table 7.7c</td>
<td>Level 2 decision table for ‘RedShoe’</td>
<td>141</td>
</tr>
<tr>
<td>Table 7.8</td>
<td>Reduced top-level decision table for ‘RedShoe’</td>
<td>141</td>
</tr>
<tr>
<td>Table 7.9</td>
<td>Cause–effect truth table for Identity</td>
<td>145</td>
</tr>
<tr>
<td>Table 7.10</td>
<td>Cause–effect truth table for NOT</td>
<td>145</td>
</tr>
<tr>
<td>Table 7.11</td>
<td>Cause–effect truth table for AND</td>
<td>145</td>
</tr>
<tr>
<td>Table 7.12</td>
<td>Cause–effect truth table for OR</td>
<td>146</td>
</tr>
<tr>
<td>Table 7.13</td>
<td>Cause–effect truth table for NAND</td>
<td>146</td>
</tr>
<tr>
<td>Table 7.14</td>
<td>Cause–effect truth table for NOR</td>
<td>146</td>
</tr>
<tr>
<td>Table 7.15</td>
<td>Reduced, extended-entry decision table for ‘RedShoe’ special</td>
<td>152</td>
</tr>
<tr>
<td>Table 8.1</td>
<td>Use case ‘Enter/Leave Building Zone’</td>
<td>158</td>
</tr>
<tr>
<td>Table 8.2</td>
<td>Basic flow for ‘Enter/Leave Building Zone’ use case</td>
<td>159</td>
</tr>
<tr>
<td>Table 8.3</td>
<td>Alternative flow for ‘Enter/Leave Building Zone’ use case</td>
<td>160</td>
</tr>
<tr>
<td>Table 8.4</td>
<td>Exception flow for ‘Enter/Leave Building Zone’ use case</td>
<td>160</td>
</tr>
<tr>
<td>Table 8.5</td>
<td>‘Buy Tickets’ use case</td>
<td>164</td>
</tr>
<tr>
<td>Table 8.6</td>
<td>Basic flow for the ‘Buy Tickets’ use case</td>
<td>165</td>
</tr>
<tr>
<td>Table 8.7</td>
<td>Exception flow for the ‘Buy Tickets’ use case</td>
<td>166</td>
</tr>
<tr>
<td>Table 8.8</td>
<td>Equivalence partitions for the ‘Buy Tickets’ use case</td>
<td>168</td>
</tr>
<tr>
<td>Table 8.9</td>
<td>Use case tests for TVM</td>
<td>168</td>
</tr>
<tr>
<td>Table 8.10</td>
<td>Extended use case test tests for TVM</td>
<td>169</td>
</tr>
</tbody>
</table>
Table 9.1  Diff-pair example for five parameters having two values each  
Table 9.2  Diff-pair-4 example for five parameters having two values each  
Table 9.3  Diff-pair-3 tests for the ‘Configuration testing’ example  
Table 9.4  ‘Configuration testing’ example: base choice  
Table 9.5  All-pairs tests for the ‘Configuration testing’ example  
Table 9.6  Comparison of some combinatorial methods  
Table 9.7  Covering array for $2^1 \times 3^1 \times 4^2$  
Table 9.8  Tests for the ‘Packaging and delivery service’ example  
Table 9.9  Base choice tests for the ‘Packaging and delivery service’ example  
Table 9.10  Diff-pair tests for the ‘Packaging and delivery service’ example  
Table 9.11  Pairwise tests for the ‘Online shop’ example  
Table 9.12  Changed tests for the ‘Online shop’ example  
Table 9.13  Diff-pair tests for the ‘Online shop’ example  
Table 9.14  Test design for the TVM example  
Table 12.1  Manual test cases for ‘Price reduction’  
Table 12.2  Test results for ‘Price reduction’  
Table 12.3  Test results for ‘Price reduction’ after feature modification  
Table 12.4  Modified test cases for RAP  
Table 14.1  Risk items for TVM  
Table 14.2  Test cases for the whole TVM  
Table 14.3  Test cases for normal mode  
Table 14.4  Test cases for reduced mode  
Table 14.5  Comparison of the test design techniques  
Table D.1  Equivalence partitions for solving Exercise 5.1  
Table D.2  Equivalence partitions for solving Exercise 5.1 – an alternative  
Table D.3  BVA for solving Exercise 5.2  
Table D.4  Decision table for solving Exercise 7.1  
Table D.5  Decision tables for solving Exercise 7.2  
Table D.6  Use case for solving Exercise 8.1  
Table D.7  Diff-pair tests for solving Exercise 9.1  
Table D.8  Pairwise testing for Exercise 9.2
István Forgács PhD was originally a researcher at the Hungarian Academy of Sciences. He has had more than 25 scientific articles published in leading international journals and conference proceedings. He is the co-author of the book *Agile Testing Foundations: An ISTQB Foundation Level Agile Tester guide*. His research interests include test design, agile testing, model-based testing, debugging, code comprehension, and static and dynamic analysis. He invented and published a partly automated method the implementation of which efficiently solved the Y2K problem so that a single tester was able to review 300,000 lines of code per day. As a result, he left academic life in 1998 to be a founder of Y2KO, the startup company implementing this project.

He is also the founder and Chief Executive Officer of 4TestDev and is a former CEO of 4D Soft. As such he led several projects funded by the European Union and others financed by national grants, where he worked together with large companies such as CERN, the European Space Agency, the Food and Agriculture Organization and the Fraunhofer Institute. He is a member of the Agile Working Group of the International Software Testing Qualification Board (ISTQB) and of the Hungarian Testing Board.

Dr Forgács is the creator of the 4Test method and initiator of the related tool Harmony, which is a test design and scriptless test automation platform for continuous testing and DevOps. He gives successful test design and model-based hands-on training.

Attila Kovács is currently a professor at Eötvös Loránd University, Budapest, Faculty of Informatics. His research interests are in software engineering, software quality, number theory and cryptography. He is the author of numerous scientific publications. He received an MSc in computer science and mathematics and a PhD in informatics. His teaching activity covers a wide range of subjects, including several courses of mathematics and software engineering.

He spent three years at the University of Paderborn, Germany, and four months at the Radboud University, Nijmegen, the Netherlands. He received an award for outstanding research from the Hungarian Academy of Sciences (Young Scientist Prize) in 1999. He is a project leader of several research and development projects, and consults for companies on issues including software quality, software testing, safety and reliability. He is an ISTQB and International Requirements Engineering Board trainer, and a founding member of the Hungarian Testing Board. He was the Program Chair of the 2016 Hungarian Software Testing Forum, the largest industrial software testing conference in Central Europe.
Are you a serious software tester? A software tester who wants to know not only how to select and design effective and efficient tests but who also wants to know why those tests work? A software tester who is ready to take a deep, intensive dive into the topic? A software tester who knows that you can only learn test design by practising test design, using realistic exercises? If so, this book is for you.

Forgács and Kovács deliver a masterful tour through the most important behavioural test design techniques. Each technique is explained thoroughly. Each technique is illustrated with examples, which is always necessary with complex topics. Further, they include code segments to illustrate exactly why these techniques work, showing specific defects the resulting tests will catch.

Because of their willingness to drill all the way from the technique through the tests to the specific defects in the code, this book will appeal beyond professional software testers. The field of testing, and who is interested in testing, has expanded quite a bit, thanks to the advance of concepts such as Agile and Lean methods, DevOps, continuous integration and continuous delivery, and continuous deployment. So, if you are a technical tester, a software development engineer in test, or a developer who cares about properly testing their code, this book is for you.

Just like writing good code, designing good tests requires hands-on practice to learn. This book provides an excellent, realistic example system to use as the basis for such exercises. Forgács and Kovács deliver a system that is universally understandable, that of a ticket vending machine. Better yet, their ticket vending machine successfully steers between the Scylla and Charybdis of such an exercise basis, avoiding being on the one hand so trivial as to be useless for learning or on the other hand being so complex as to distract from the techniques being taught. From my own experience writing books on test design and creating courses on test design, I know how hard it is to create a good basis for exercises, and how critical it is to do so.

So, if you have reached a point with your knowledge of software testing where you need to challenge yourself, to gain deeper understanding, to become ready to put sophisticated test design techniques into practice, then, I’ll say it again, this book is for you. Invest the time to read the explanations of the techniques, carefully study the examples, and work through the exercises. Your return on that investment will be a significant step forward in your software testing skills and abilities.

Rex Black
President, RBCS Inc.
Past President, ASTQB
Past President, ISTQB

xiv
This is the testing book I’ve been waiting for. It’s nice that the authors understand and include some testing theory, but what sets this book apart in my mind is that it is written by practitioners who also have real pragmatic testing experience and a deep understanding of important issues. Also of critical importance in these days of ubiquitous embedded systems and the internet of things (IoT), is that the authors recognise and explicitly state issues related to hardware that also have to be considered. For example, in their running example of the ticket vending machine, they mention various types of testing that would have to be included, related to the physical machine. I’ve found that most students, and even many practitioners, just assume that the only possible problems that need to be considered are ones related to the code itself.

Other nice features are the device the authors use at the beginning of each chapter, a brief summary of what is to come (‘Why is this chapter worth reading?’) as well as a brief wrap-up that comes at the end of the chapter (‘Key takeaways’); and that each chapter has non-trivial examples and exercises.

Finally, it is really important that the authors have included issues related to testing systems developed using modern software development strategies such as Agile and DevOps.

This is a book that both students and practitioners can use to hone their software testing skills.

Elaine Weyuker
University Distinguished Professor
Member, US National Academy of Engineering, ACM Fellow, IEEE Fellow
ACKNOWLEDGEMENTS

We are grateful to all of those whom we have had the pleasure to work with during this book-writing project: colleagues, friends, specialists.

First and foremost, we thank our editor for her professionalism and strenuousness. Her feedback on our book was thoughtful, thorough and supportive.

Second, we acknowledge all of our reviewers. They did an excellent job: Péter Földházi, Zsolt Hargitai, Terézia Kaukál, Zoltán Micskei, Gábor Árpád Németh, Ceren Sahin and Kristóf Szabados. Grateful thanks go to the official reviewers as well. Any remaining errors and omissions are the authors’ responsibility.

Great appreciation is given to the crew at the Hungarian Testing Board for their continuous support.

We are grateful to Bianka Békefi for implementing our key example, the ticket vending machine (TVM), and for reviewing the manuscript very carefully. Special thanks to József ‘Joe’ Csuti, for seeding tricky bugs into the TVM code, even if we could find them. We are also grateful for László Czippán who helped us with some coding. Many thanks to Dalma Bartuska who made exploratory testing for the TVM.

Attila sends special thanks to his department leader, Péter Burcsí, who granted him peace in the book-writing period.

Nobody has been more important to us in the pursuit of this project than the members of our families. We acknowledge them for the continuous and strong support they gave us.

István: this almost two-year journey would not have been possible without the support of my family. Thank you to Margó, my wife, for encouraging and inspiring me all the time to follow my dreams. I am especially grateful to my parents, who have supported and believed in me my whole life.

Attila: I would like to thank my family members for their patience and continuous support, especially my sons, Bendegúz and Barnabás, my love and my parents who always believed in me.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>acceptance criteria</td>
</tr>
<tr>
<td>AETG</td>
<td>automatic efficient test generator</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>BDD</td>
<td>behaviour-driven development</td>
</tr>
<tr>
<td>BPMN</td>
<td>business process modelling notation</td>
</tr>
<tr>
<td>BVA</td>
<td>boundary value analysis</td>
</tr>
<tr>
<td>CD</td>
<td>continuous deployment/delivery</td>
</tr>
<tr>
<td>CI</td>
<td>continuous integration</td>
</tr>
<tr>
<td>CPH</td>
<td>competent programmer hypothesis</td>
</tr>
<tr>
<td>CT</td>
<td>classification tree</td>
</tr>
<tr>
<td>DAG</td>
<td>directed acyclic graph</td>
</tr>
<tr>
<td>DFA</td>
<td>deterministic finite automaton</td>
</tr>
<tr>
<td>DP</td>
<td>defect prevention</td>
</tr>
<tr>
<td>EFSM</td>
<td>extended finite-state machine</td>
</tr>
<tr>
<td>EP</td>
<td>equivalence partitioning</td>
</tr>
<tr>
<td>ET</td>
<td>exploratory testing</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
</tr>
<tr>
<td>FDD</td>
<td>feature-driven development</td>
</tr>
<tr>
<td>FSM</td>
<td>finite-state machine</td>
</tr>
<tr>
<td>GDPR</td>
<td>General Data Protection Regulation</td>
</tr>
<tr>
<td>GSM</td>
<td>global system for mobile communication</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>GW</td>
<td>GraphWalker</td>
</tr>
<tr>
<td>HRF</td>
<td>high-risk function</td>
</tr>
<tr>
<td>IoT</td>
<td>internet of things</td>
</tr>
<tr>
<td>ISTQB</td>
<td>International Software Testing Qualification Board</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>KDT</td>
<td>keyword-driven testing</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>LOC</td>
<td>lines of code</td>
</tr>
<tr>
<td>LRF</td>
<td>low-risk function</td>
</tr>
<tr>
<td>MBT</td>
<td>model-based testing</td>
</tr>
<tr>
<td>OA</td>
<td>orthogonal array</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>SBE</td>
<td>specification by example</td>
</tr>
<tr>
<td>SBTM</td>
<td>session-based test management</td>
</tr>
<tr>
<td>SDET</td>
<td>software development engineer in test</td>
</tr>
<tr>
<td>SDL</td>
<td>specification and description language</td>
</tr>
<tr>
<td>SDLC</td>
<td>software development life cycle</td>
</tr>
<tr>
<td>STT</td>
<td>state transition testing</td>
</tr>
<tr>
<td>SUT</td>
<td>system under test</td>
</tr>
<tr>
<td>T</td>
<td>true</td>
</tr>
<tr>
<td>TAC</td>
<td>test data adequacy criterion</td>
</tr>
<tr>
<td>TCP</td>
<td>test case prioritisation techniques</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>TDD</td>
<td>test-driven development</td>
</tr>
<tr>
<td>TFT</td>
<td>thin film transistor</td>
</tr>
<tr>
<td>TSC</td>
<td>test selection criterion</td>
</tr>
<tr>
<td>TTCN-3</td>
<td>Test and Test Control Notation Version 3</td>
</tr>
<tr>
<td>TVM</td>
<td>ticket vending machine</td>
</tr>
<tr>
<td>UI</td>
<td>user interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>verification and validation</td>
</tr>
</tbody>
</table>
Black-box testing: Functional or non-functional testing without referencing the internal structure of the component or system.

Boundary value analysis: A black-box test design technique in which test cases are designed based on boundary values of equivalence partitions.

Business rule-based testing: A black-box test design technique used to determine test scenarios and test cases for business logic.

Cause–effect graph: A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

Classification tree testing: An approach of equivalence partitioning that uses a descriptive tree-like notation. The equivalence partitions are hierarchically ordered.

Combinative testing: A black-box test design technique in which the number of tests can be described by a linear function of the maximal number of parameter values.

Combinatorial testing: A black-box test design technique in which test cases are designed to execute specific combinations of values of several parameters.

Competent programmer hypothesis: Programmers create programs that are close to being correct, i.e. the specification and the appropriate code are close to each other.

Decision table testing: A black-box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a table.

Defect prevention: A process that identifies, analyses and collects defect information in order to prevent implementing new defects in the code.

Diff-pair testing: A combinatorial test design technique that requires each value of any parameter to be tested with at least two different values for any other parameters.

Domain-based testing: A black-box test design technique that is used to identify efficient and effective test cases when more parameters can or should be tested together. It establishes, builds on and generalises equivalence partitioning and boundary value analysis.
Equivalence partitioning: A black-box test design technique in which test cases are designed to exercise disjoint partitions by using at least one representative input value of each partition. If a single input detects (or not) a failure then the system behaves the same way for all other inputs in the same partition.

Exploratory testing: An approach to testing whereby the testers dynamically design and execute tests based on their knowledge, exploration of the test item and the results of previous tests.

Model-based testing: Testing based on or involving models.

Mutation testing: A method of testing the tests, in which slightly modifying the original code creates several mutants. A reliable test data set should then differentiate the original code from the well-selected mutants.

On-the-fly testing: An approach to testing where test cases are created on the fly and executed immediately based on the running application.

Orthogonal array: A two-dimensional array constructed with special mathematical properties. Namely, there is an integer t so that for every selection of t columns of the table, all ordered t-tuples of the symbols, formed by taking the entries in each row restricted to these columns, appear the same number of times.

Risk analysis: The overall process of risk identification and risk assessment.

Scenario-based testing: A test design technique focusing on the interactions between an ‘actor’ and the system, with different roles and environments.

Session-based testing: A software test method that aims at combining accountability and exploratory testing in order to provide fast defect discovery, creative on-the-fly test design, management control and reporting metrics.

State transition testing: A black-box test design technique in which the behaviour of an application under test is analysed for different input events in a sequence. The technique models the specification via states and transitions, which are covered according to a test selection criterion. In this technique both the valid and invalid transitions are checked.

Test case: A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

Test data adequacy criteria: Rules for validating whether enough testing has been executed.

Test design: The activity of deriving and specifying test cases from test conditions.

Test path: An execution path showing the control flow when executing a test case.

Test script: A sequence of instructions for the execution of a test.
Test selection criterion: The criteria used to guide the creation of test cases or to select test cases in order to limit the size of a test.

Test suite: A set of test cases or test procedures to be executed in a specific test cycle.

Use case testing: A black-box test technique in which test cases are designed to execute scenarios of use cases.

User story: A high-level user or business requirement commonly used in Agile software development to capture a software feature from the end-user point of view. It typically consists of one sentence in the everyday or business language and acceptance criteria for validation.

White-box testing: Testing based on an analysis of the internal structure of the component or system.
Dear Reader,

Thank you for picking up our book. This book is about specification-based test design.

We promise that you will learn a lot, and that after having read it, you will be a much better tester than before. The book not only assists in the deeper understanding of software testing but also strongly supports preparation for various professional exams (for example, those of ISTQB).

Each chapter is self-explanatory, and we refer to the necessary parts of each for all needs. Each chapter includes at least two elaborated examples. There is (at least) one for demonstrating the described method, and another, which presents the method through a common example. At the end of each chapter, there are exercises to be solved. We strongly suggest doing these exercises. You can find their solutions in Appendix D.

The entire book contains useful information, although we separate the must-read and worth-reading parts out. The worth-reading parts are denoted by a book icon in the margin; however, you can solve all the exercises without reading these parts.

A hint/tip is denoted throughout by a magnifying glass icon in the margin.

An anecdote is denoted throughout by a speech bubble icon in the margin.

Happy reading,
I. Forgács and A. Kovács
Would you think, dear reader, that testing is easier than developing code? Some people think ‘Yes, of course’, since programming is hard, and requires a lot of skill to be able to do it effectively. Testing, on the other hand, is just monkey typing and watching the results, right? Other people think ‘No, by no means’, testing is hard, and excellent testing is far more difficult than excellent engineering, since excellent software testing requires thinking out of the box, requires selecting and applying test design techniques in combination, and requires intuition for finding tricky bugs. Others think that it depends at least on complexity, which is probably the most common answer.

Probably all of you know a test automation engineer, a recognised superman or superwoman, who incorporates the knowledge of developers and testers. However, non-testers sometimes believe that testing requires only some logical thinking and that is all. Test job applicants might think that learning some basic test design methods, a bug tracking system, and a version control system is enough for a testing career. The sad truth is that it is not enough. Testing is not only designing and executing some test code. A tester is an architect, a designer and far more. Testing is an art, a craft, a discipline. However, there’s still a long road to traverse before testing occupies a worthy place in the engineering universe.

When the former company of one of the authors of this book was looking for testers, lots of non-IT people applied for the available positions. Some were economists, lawyers and there was even a policeman. Most applicants believed that a tester’s job is similar to a soccer referee’s job – everybody knows how to do it. There were also positions for Java developers and in that case, all the applicants were Java developers.

The good news is that this book is for everyone who wants to know why testing is complex, and in many cases, why it is even more complex than coding. Here we try to demonstrate some aspects of this.

**STARTING POINT**

Our starting point is a specification containing requirements on how to optimally arrange several rectangular and square shapes in a minimal quadrant. With this, we demonstrate the advantages of conscious test design and describe the usefulness of
implementation-independent testing. This example served as an exercise for tester applicants in a company. Everybody who was able to solve this problem, later became a good tester. The applicant’s task was to figure out a single test case, that if the implementation of the specification was faulty, then there was a high probability that the test would detect the bug.

SPECIFICATION: SQUARE-FILLING

For this example, applicants were given three kinds of rectangular shapes with their length and height, A = 1x1, B = 2x1 and C = 2x2.

Rectangle B is not allowed to rotate into the rectangle 1x2. The input is a finite random list of these rectangular shapes containing at least one shape from each type.

Example

Input:

1x1 – 3 pieces,
2x1 – 4 pieces,
2x2 – 1 piece.

The task is to insert the objects into a square with the smallest possible size.

A possible output

Clearly, to create the appropriate test case, the tester should know something about the potential solution, which involves the following steps:

1. Summarise the area of the shapes.
2. Compute the square root R of the total area.
3. If R is an integer, then this can be the size of the minimum square.
4. If R is a non-integer, then let’s select the closest integer S for which S > R. Then S can be the size of the minimum square.
5. Starting from the top left of the minimum square, let’s place the 2x2 squares from left to right. When a row is full, we continue the placement of the shapes in the next row beneath the full one.
6. If there are no 2x2 shapes, then we place shapes 2x1 in the same way.
7. Finally, we place all the remaining 1x1 shapes into the empty parts of the minimum square.
8. If the shapes do not fit into the minimum square, then increase S by one, and
9. Place the unplaced shapes into the empty space in the way described in steps 5-7.

Then, the tester has to select an order of the input shapes. Even if the tester knows the above method, it is not easy to create the appropriate test. To create a test for which a faulty solution will most probably fail, the tester has to know something more. Really, the tester also has to know that squares with odd sizes are more difficult to fill in. The reason is that if the size is odd and we start filling the minimum square with 2x1 shapes as described in step 5, then some 2x2 shapes cannot fit. This knowledge is necessary for the tester but not necessary for the developer.

Consider the test case design. If the input forms are in the order in which the minimum squares are filled only by selecting the following form and inserting them into the first empty space, the test case is ineffective. An odd-sized minimum square is more challenging for testing than an even-sized one, as already mentioned.

**EXAMPLE: 7X7 SQUARE**

Input:

1x1 – 7 pieces,

2x1 – 7 pieces,

2x2 – 7 pieces.

Since the area is 7+14+28 = 49, the size of the minimum square is 7. The input can be: T = [1x1, 1x1, 1x1, 2x2, 2x1, 2x1, 2x2, 1x1, 2x2, 2x2, 2x1, 1x1, 1x1, 1x1, 2x2, 2x1, 2x2, 2x1, 2x1, 2x2, 1x1, 2x2, 2x1, 2x1]. The output is the following. The numbers represent the order of insertion.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>

The author gave the above input to all of the interviewees after they had come up with their solutions. They immediately realised the problem with their solution when they were not able to insert the rectangular shapes into the 7x7 square. This test case always worked.
There are some other questions to consider:

a. should the objects fill the minimum square entirely? or
b. leave some empty places in the square? or
c. should the total area of the objects just exceed a square?

Of course, the last option is the worst. Namely, a wrong solution may result in a possible arrangement, since we have lots of space to fill. The less extra space is available, the greater the chance that a faulty solution fails. Therefore, the first is the best selection for a reliable test.

There is another area where testers have difficulties. A developer always knows when they are finished. When the implemented code is working, and the unit tests have passed then the developer gives the code to testers for further testing. However, it is not obvious when the tester is finished: how many test cases need to be designed and executed? The answer is: it depends.

If the goal is to test the suitability of an applicant for a job in the company, then this one single test is enough. Through this we learn if the applicant is able to solve the problem and implement their solution. However, if the task is to test a larger system and faulty implementations may cause serious problems, then additional test cases need to be created. For example, we have to test the size of the minimum square. It means that we have to select a suitable test selection criterion to restrict the number of test cases.

Our simple square-filling example shows that testing can be as complex as coding. Software complexity grows to the limits of our ability to manage that complexity. Testing complex software systems is one of the biggest challenges in our engineering life. Developers ensure that the software meets the specifications, business owners ensure that software systems meet the business’s operational needs, and testers provide them with information about the quality. To reach the main goal of the software development process, which is a high-quality output, one will have to plan, design and conduct reliable tests.

There are various types of test design techniques, each of which is suitable for identifying a certain type of defect. Therefore, the challenge lies in selecting the right set of relevant test design techniques for the particular application. By designing appropriate test cases, the complexity of the quality assurance costs and time can be reduced. However, in practice, the known techniques are applied together, which raises the complexity and difficulty of the test design. This book deals with the question of how to manage these issues.

There are wonderful books and papers on software test design (see the references at the end of this book). In this book, we not only describe and explain the different methods of test design but always show why to use and how to use them by applying a realistic software specification. Also, the book always shows alternative solutions and points out why something is inappropriate. Many examples in the book are based on an implemented version of a software specification, therefore ‘real faults’ can be detected.
The book explains and evaluates the most important and applicable methods generally and specifically, unfolding their fault-finding capabilities. The different test design techniques will be connected and applied together. We also present the theoretical background of the techniques and their relations to real-world solutions. The presented examples are complex but do not require any special domain knowledge. The book covers all the usable test design techniques and the ways to validate them; it contains new methods, ideas and test selection criteria.

THE STRUCTURE OF THE BOOK

This book consists of 15 chapters and four appendices. The main chapters contain the test design techniques following a common structure. We start with the description of the method and demonstrate it by applying simple examples. We then may show complex examples as well. Then we select an appropriate part of our key example, the ticket vending machine (TVM), to also demonstrate the technique. The next part is the method evaluation, describing the types of defects, applicability, advantages and shortcomings of the technique. For interested readers, we present a short section of theoretical background. We summarise the chapters with key takeaways, followed by some exercises.

The first chapter introduces and summarises the whole testing process, placing the test design into it. We also consider some basic methods, which are necessary for test design.

In Chapter 2, you’ll find our basic example, the TVM, which is implemented in Java. We use this example throughout the book.

In every test design technique chapter, we select a reasonable part of the TVM for demonstration. In the last chapter, we test the whole application. As the implementation of TVM was too good (thanks to our developer Bianka), we asked our experienced Java architect to seed some tricky bugs into it.

In Chapter 3, we deal with risk management and test optimisation. You will learn how to optimise testing to minimise the total system development life cycle costs based on risk and complexity analysis and by applying an appropriate set of test design techniques.

In Chapter 4, we demonstrate the importance of defect prevention, a cheap and reliable method of avoiding defects, and show you how to perform it.

Following these chapters, which describe how to support and complete test design, we consider all the important traditional test design techniques.

Chapter 5 contains domain-based techniques. You will learn about equivalence partitioning, boundary value analysis and domain analysis from more different perspectives than in other books. We will show you an optimal test selection criterion, which will be used later in our examples.
In Chapter 6 we deal with state transition testing. You will learn about state transition diagrams and tables. We present interesting examples for demonstration and introduce new test selection criteria. We show how state transition testing, equivalence partitioning and boundary value analysis can be used together in the case of more complex specifications.

In Chapter 7 you will learn business rule-based test design techniques. Specifically, we deal with decision table testing and cause–effect graphs. Decision tables guarantee that every possible combination of condition value is considered.

Chapter 8 is about scenario-based testing. We start with use case testing, which is used widely in developing tests at the system or acceptance level. Then we follow with user story testing, and we introduce Gherkin syntax.

Chapter 9 is about combinative and combinatorial testing. Combinative testing requires only a linear number of test cases, yet these tests may detect bugs caused by different issues in parallel. Here we introduce a new method, diff-pair testing. Then, we describe the combinatorial methods and we show you the strengths and the weaknesses of them.

In Chapter 10 you will learn about exploratory and session-based testing. We apply them for testing the whole TVM containing natural and artificial bugs and we show you the results.

In the next chapters, we move from traditional test design to automated test case generation, that is model-based testing (MBT).

In Chapter 11 we provide an overview of MBT in general and show you why it is promising.

In Chapter 12, you will become acquainted with Gherkin-based MBT. We will show you how to prepare a model through iterations, and how to make models for complex cases when more test design techniques are used together.

Chapter 13 is perhaps the most practical chapter, where we introduce you step-by-step to the GraphWalker, a graphical MBT tool. We demonstrate random test execution as an extension to the test design techniques.

In Chapter 14 we test the whole TVM by applying multiple techniques and compare the results. You will learn how many natural and seeded defects are found by applying different techniques and test selection criteria.

The final part is Chapter 15 with conclusions and recommendations.
1 INTRODUCTION

WHY IS THIS CHAPTER WORTH READING?

This introductory chapter explains the importance of software testing, and covers what test design is, how we can put test design activities into the software development and testing life cycle, why tests should be designed, what factors affect test design and when test design should be done.

THE IMPORTANCE OF SOFTWARE TESTING

Software is a vital part of our everyday life. It helps in navigating to destinations, communicating with people, driving production, distributing energy resources, taking care of people’s health, and so much more. We use software for entertainment, for scientific research, for driving companies forward. There is embedded software in cars, trucks, locomotives, aeroplanes, mobile phones, electronic equipment and so on. Controlling and ensuring the quality of these software systems is vital. We need to check, that is test, software because things can always go wrong – humans make mistakes all the time. Despite this, the human aspects of producing quality software are indisputable even if artificial intelligence comes to the fore.

ISTQB certifies software testers worldwide; there are books, articles, documents, webinars, and blogs supporting the knowledge transfer of testing. In this book, we mainly draw on the terms of the ISTQB Glossary (2018a).

Let’s list a few goals that should be considered while testing software systems:

- ensuring the overall quality of the systems;
- customer satisfaction;
- reducing the risk of failures;
- cost-effective maintenance.

The most important goal of testing is to ensure good quality software by optimising the project costs in a way that all the parties involved gain confidence in the product – see
Chapter 3. To be able to do this, a tester has to harvest information on system behaviour. One of the main reasons for gathering information is the execution of test cases. The test case is an essential notion in software testing: simply, *a test case is a set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed on test conditions* (ISTQB, 2018a). At this point the main questions are:

- How do we design the tests?
- Which test cases are the most appropriate for a given situation?
- How can test intensity be determined?
- How many test cases do we need?
- How can we validate the tests?

In answering these questions the process of **test design** is essential.

**WHAT IS TEST DESIGN EXACTLY?**

Test design is one of the most important prerequisites of quality. We design tests to support:

1. defining and improving quality-related processes and procedures (quality assurance);
2. evaluating the quality of the product with regards to customer expectations and needs (quality control);
3. finding defects in the product (testing).

Fortunately, or unfortunately, it is a creative process on its own, but also one that requires technical expertise.

More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded - indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

(Beizer, 1990)

Test design depends on many factors. **Figure 1.1** summarises the relevant entities of the traditional testing life cycle including test design activities.

**Preceding test design activities – test planning and test analysis**

In this subsection we describe the main activities that precede and influence test design.
Test planning

The test planning process determines the scope, objective, approach, resources and schedule of the intended test activities. During test planning – amongst others – the test objectives, test items, features to be tested, the testing tasks, human and other resources, the degree of tester independence, the test environment, entry and exit criteria to be used, and any risks requiring contingency planning are identified.


A test policy document represents the testing philosophy of the company, selecting the frames that testing parties should adhere to and follow. It should apply to both new projects and maintenance work. The test strategy is defined as a set of guiding principles that influences the test design and regulates how testing is carried out.

The test approach defines how (in which way) testing is carried out, that is how to implement a test strategy. It can be proactive, that is the test process is initiated as early as possible to find and fix the faults before the build (preferable, if possible), or reactive, which means that the test design process begins after the implementation is finished. The test approach can be of different (not necessarily disjoint) types such
PRACTICAL TEST DESIGN

as specification-based, structure-based, experience-based, model-based, risk-based, script-based, fault-based, failure-based, standard-compliant, test-first and so on, or a combination of them. For example, a risk-based approach is applied in almost all situations.

The proactive test-first approach starts with analysing the specifications, then applying risk analysis. Based on the result of this analysis one can select the appropriate techniques to design the test cases. The riskier code needs more techniques to be used in parallel. The test design can then be extended with structure-based testing, that is coverage analysis to create additional test cases for uncovered code.

**Test monitoring and control**
Test monitoring is an ongoing comparison of the actual and the planned progress. Test control involves the actions necessary to meet the objectives of the plan.

Although monitoring and control are activities that belong to the test manager, it is important to ensure that the appropriate data/metrics from the test design activities are collected, validated and communicated.

**Test analysis**
The test engineering activity in the fundamental test process begins mainly with test analysis. Test analysis is the process of looking at something that can be used to derive quality information for the software product. The test analysis process is based on appropriate project documents or knowledge, called the test basis, on which the tests are based.

The test analysis phase has three main steps before the review.

1. The first step is to **analyse the test basis thoroughly**. Possible test bases can take the form of business requirement documents, systems requirement documents, functional design specifications, technical specifications, user manuals, source codes and so on.

2. The second step is to **identify the test objects (features, scenarios) and conditions** by defining what should be tested.

   **A test condition** is a statement about the test object, which can be true or false. Test conditions can be stated for any part of a component (or system) that could be verified by some tests, for example a function, transaction, feature, quality attribute or structural element.

   **A feature** is an attribute of a component or system specified or implied by requirements documentation.

   **A test scenario** is a business requirement to be tested, which may have one or more tests associated with it.

The advantages of using test scenarios are:

a. They help to achieve more complete test coverage.

b. They can be approved by various stakeholders.

c. They help to determine the end-to-end functionality of the test object.
Projects that follow Agile methodology – like Scrum or Kanban – usually do not use test scenarios: they apply user stories, sometimes use cases. We note that both are test objects, and a user story – which is an informal, natural language description of some system features – contains information about the test conditions.

A test object may consist of different test items. A test item is a specific element to be tested.

**EXAMPLE FOR TEST OBJECT AND TEST ITEM**

To clarify the notion, a test object can be login functionality, while a test item can be a username or password field in the login form. A test condition for this scenario can be an assertion for usernames such as ‘a username shall start with an alphabetical or numeric value’.

It is usually advisable to determine the test objects/conditions at different levels of detail. Initially, high-level objects/conditions have to be identified to define general targets for testing. Subsequently, in the design phase, the objects/conditions will be groomed towards specific targets. Applying this type of hierarchical approach can help to ensure that the coverage is sufficient for all high-level items.

3. The third step of the test analysis phase is risk analysis. For each elicited (mainly high level) test object the risk analysis process determines and records the following risk attributes:

   a. the impact of malfunctioning (how important the appropriate functioning of the test is);
   b. the likelihood of malfunctioning (how likely it is to fail).

Later, during test execution, the risk inventory supports the organisation of the tests in a way that the high-risk items are addressed with more intensive testing – see Chapter 3.

An effective risk analysis is a result of contributions from different stakeholders: managers, technical and business people, customer representatives and so on. Risk analysis is a continuous process.

We note that risk analysis is always necessary, not only in risk-based testing. The reason is that we should optimise the costs and effort of the whole test development process. Risk analysis performed during the test analysis phase refines the risks identified during test planning.

4. The last step of this phase is spent validating the identified artefacts, test objects, main scenarios and test conditions, and the risk inventory by applying reviews.
Present activity – test design

After the test analysis, the **test design** comes into the picture. The test design depends on a few factors:

- The result of risk and complexity analysis – see Chapter 3. This is the starting point of the test design techniques to be selected.
- The way the software development process is organised (for instance, V-model vs Agile).
- The knowledge and experience of the people involved.
- The available (sometimes limited) resources.

Based on these and other factors, the test design can be performed in seven steps (see Figure 1.1):

1. First, the **test approaches** are worked out, which means that it should be planned how the product quality together with the cost optimisation goals can be achieved via the approaches defined earlier, taking into consideration the elicited test conditions, code complexity and risks. Here we consider consistencies with standards, processes, flows, measures, user experience and so on.

2. After planning the test approach, the **test design techniques** are selected that meet the testing objectives and the result of risk and complexity analysis. In general, it is advisable to select test design techniques understandable to other stakeholders. Real systems usually require using more than one test design technique together. Table 1.1 shows some test techniques related to different test design categories and test bases.

<table>
<thead>
<tr>
<th>Test basis</th>
<th>Test design category</th>
<th>Test design technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal requirements</td>
<td>Specification-based</td>
<td>Equivalence partitioning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boundary value analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decision table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cause and effect graph</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classification tree</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ad hoc testing</td>
</tr>
<tr>
<td>Semiformal or formal</td>
<td>Model-based,</td>
<td>State transition testing</td>
</tr>
<tr>
<td>requirements</td>
<td>Model checking</td>
<td>Syntax testing</td>
</tr>
</tbody>
</table>

(Continued)
3. The next step is to determine the test case selection criteria (for simplicity we refer to this as test selection criteria).

First, let’s consider the types of test cases. When a test case checks valid functionality then it is called a **positive test case** (or sometimes ‘happy path’); when a test case checks invalid functionality, then it is called a **negative test case**. Both positive and negative testing are equally important for quality software. Negative testing helps to find more defects, but it is done once positive testing is completed.

Usually, we distinguish **high-level** (abstract) and **low-level** (concrete) test cases. An abstract test case provides guidelines for the test selection using tables, lists and graphs. A concrete test case realises the abstract one by describing the preconditions, test steps, expected results, postconditions and by assigning concrete input values to it. Sometimes, the level of detail is forced by an outside authority (DO 178-C in avionics, for example). A good test case satisfies at least the following:

- **Accurate**: has a clear purpose.
- **Effective**: able to verify correctness or able to uncover faults depending on the test target.
- **Economic**: no unnecessary steps, cheap to use.
- **Evolvable**: easy to maintain.
- **Exemplary**: has low redundancy.
- **Traceable**: capable of being traced to requirements.
- **Repeatable**: can be used to perform the test several times.
- **Reusable**: can be ‘called’ from other test cases, avoiding duplication and therefore the room for human error.
A TEST CASE AS AN ‘EXAMPLE’

A test case can be seen as an example for the specification. Examples are very useful measures of understanding anything. They can also be used to understand software specification, user stories and so on. In most cases, a test case is a good example. For example, if we test a login feature with test data, we can see that it is also an example for demonstrating some part of the login process.

Good test cases are understandable examples for any stakeholders. The project owner can validate the specification or user story; the developers can implement better code because of a deeper understanding. The tester is supported by the examples and gets feedback from the examples/test cases. Moreover, and this is also important, if test design has been done, we do not omit the test.

In agile testing, the Specification by Example (Adzic, 2011) uses this approach, that is the Agile team creates good examples, which can be considered as test cases and also as part of a specification. By applying examples, the problems in the specifications/user stories can be detected very early on.

The test selection criteria determine when to stop designing more test cases, or how many test cases have to be designed for a given situation. The test selection criterion for the equivalence partition method, for example, may result in designing a different number of test cases for each equivalence class.

It is possible to define different (‘weaker’ or ‘stronger’) test selection criteria for a given condition and design technique.

EXAMPLE FOR WEAK AND STRONG TEST SELECTION CRITERIA

Assume that during test design a technique was selected for a test condition where the input domain was partitioned into 100 disjoint subdomains. A ‘weak selection criterion’ with respect to the criterion and technique means for example choosing at most one item of test data from each subdomain. A ‘strong selection criterion’ means for example choosing test data on each of the borders between the subdomains, test data ‘close to’ the borders and test data from anywhere else in each subdomain.

There is an important notion closely related to test selection criteria, namely, the test data adequacy criteria.

While test selection criteria are defined concerning the test design, that is independently of the implementation, test data adequacy criteria are defined concerning program execution. Both test selection and test data adequacy criteria, however, provide a way to define a notion of ‘thoroughness’ for test case sets. By applying these criteria, we can check whether our test set is adequate and no additional testing is needed. We can call both test selection and test data adequacy criteria with regards to thoroughness.
The adequacy criteria define 'what properties of a program must be exercised to constitute a thorough test' (Goodenough and Gerhart, 1975). Satisfying a stronger adequacy criterion implies greater thoroughness in testing. The test data adequacy criteria validate whether the designed test cases are efficient concerning the testers’ expectations, and are applied after the test cases have been executed. Test data adequacy criteria focuses on validating the fulfilment of a specific code coverage and on the creation of additional test cases to detect specific types of defects.

**THEORETICAL BACKGROUND**

Let’s formalise these notions. Consider a program $P$ written to meet a set of requirements $R = \{R_1, R_2, \ldots, R_m\}$. Suppose now that a set of test cases $T = \{T_1, T_2, \ldots, T_k\}$ has been selected to test $P$ to determine whether or not it meets all the requirements in $R$. The question arises naturally: is $T$ good enough? Or in other words, is $T$ adequate? Hence, adequacy is measured for a given test set designed to test $P$ to determine whether or not $P$ meets its requirements. This measurement is done against a given adequacy criterion $C$.

A test set is considered adequate concerning criterion $C$ when it satisfies $C$. Let $D_c$ be a finite set of sub-criteria items belonging to criterion $C$, called **coverage domain**. We say that $T$ covers $D_c$ if for each coverage item $x$ in $D_c$ there is at least one test case $t$ from $T$ that tests $x$.

The test set $T$ is considered adequate concerning $C$ if it covers all items in the coverage domain. Similarly, $T$ is considered inadequate concerning $C$ if it covers only $j$ items of $D_c$ where $j < n$ and $n$ is the number of the coverage items. The fraction $j/n$ is a measure of the extent to which $T$ is adequate concerning $C$. This ratio is also known as the coverage of $D_c$ concerning $T$, $P$ and $R$. The coverage domain may depend on the program $P$ under test or on the requirements $R$ or both.

The examples below demonstrate test selection and test data adequacy criteria and the difference between them.

**EXAMPLE FOR TEST SELECTION CRITERIA**

Regarding the test design-related aspects of test selection criteria, consider the following requirements:

- **R1.** Input two integers, say $x$ and $y$ from the standard input device
- **R2.** If $x$ and $y$ are both negative, then print $x + y$ to the standard output device
- **R3.** If $x$ and $y$ are both positive, then print their greatest common divisor to the standard output device
- **R4.** If $x$ and $y$ have opposite signs (one of them is positive and the other is negative), then print their product to the standard output device
- **R5.** If $x \times y$ is zero then print zero to the standard output device.
After analysing the requirement set, \( R = \{R_1, R_2, R_3, R_4, R_5\} \) one can see that \( R \) is consistent, does not contain redundancy and is deterministic. Let’s choose the test approach as a specification-based one. Suppose that the program \( P \) satisfies the requirement set \( R \). Let’s specify a test selection criterion (TSC) as follows:

**TSC:** a test set \( T \) for the program \( P \) is considered adequate if for each requirement \( r \) in \( R \) there is at least one test case \( t \) in \( T \) such that \( t \) tests the correctness of \( P \) with respect to \( r \).

In our case the coverage domain is \( R \) itself. With the test set \( T_1 = \{((-2, -3); -5), ((2, 3); 1)\} \) (i.e. for input \( x = -2 \) and \( y = -3 \) the expected output is -5, for input \( x = 2 \) and \( x = 3 \) the expected output is 1) then \( T_1 \) covers \( R_1, R_2 \) and \( R_3 \), but does not cover \( R_4 \) and \( R_5 \). Hence, \( T_1 \) is inadequate with respect to TSC and the coverage of \( T_1 \) is \( \frac{3}{5} = 60\% \). The test set \( T_2 = \{((-2, -3); -5), ((4, 2); 2), ((2, -4); -8), ((2, 0); 0)\} \) is adequate with regards to TSC, having 100% coverage. The appropriate test design technique for this scenario is called equivalence partitioning (see Chapter 5).

**EXAMPLE FOR TEST DATA ADEQUACY CRITERIA**

1. function coverage_demo
2. begin
3. int x, y;
4. input(x, y);
5. if x < 0 and y < 0 then
6. output(x+y);
7. else if x > 0 and y > 0 then
8. while x ≠ y
9. if x > y then
10. x = x-y
11. else y = y-x;
12. output(x);
13. else if sgn(x) * sgn(y) = -1 then
14. output(x * y);
15. else output(0);
16. end

In the pseudocode above the function *coverage_demo* is implemented. Line 3 contains the declarations, line 4 the input from the standard input device (requirement R1). Lines 5–6 contain the code for the requirement R2, lines 7–12 the Euclidean algorithm for requirement R3, lines 13–14 the code for requirement R4, and finally, line 15 contains the code for requirement R5.

Consider the following **test data adequacy criterion (TAC):**

**TAC:** a test set \( T \) for the program \( P \) (*coverage_demo*) is considered adequate if each statement in \( P \) is traversed at least once.
The coverage domain now is the set of all possible statements in P. Let the test set be $T_2 = \{([-2, -3]; -5), ([4, 2]; 2), ([2, -4]; -8), ([2, 0]; 0)\}$ as before. In the first test case $([-2, -3]; -5)$ the path 1-2-3-4-5-6-16 is executed. In the second test case $([4, 2]; 2)$ the path 1-2-3-4-5-7-8-9-10-8-12-16 is executed. In the third test case $([2, -4]; -8)$ the path 1-2-3-4-5-7-13-14-16 is executed, and in the fourth test case $([2, 0]; 0)$ the path 1-2-3-4-5-7-13-15-16 is executed. Hence, this test set is inadequate with regards to TAC. The coverage of $T_2$ is $14/15 = 93\%$ (the statement in line 11 is not covered).

Adequacy and full coverage can be achieved by $T_3 = \{([-2, -3]; -5), ([2, 3]; 1), ([2, -4]; -8), ([2, 0]; 0)\}$.

Observe the difference between test selection and test data adequacy criteria. During test design we considered R3 as a simple requirement in the previous example, which can be covered by a single test case. On the other hand, the implementation (see lines 7–12) is more complex and may require more test cases.

Table 1.2 shows some coverage domains with test selection/test data adequacy criteria. Definition is an assignment statement, such as $x = 1$. Use occurs when the variable is read from memory, such as $z = x + 1$. Definition–Use pairs exist for a definition and a use of the same variable so that there is a program path between them where no other definition for that variable occurs.

It is important to note that some authors define test selection coverage as simply the same as code coverage. However, although test selection coverage and code coverage are both useful to assess the quality of the application code,

<table>
<thead>
<tr>
<th>Coverage domain</th>
<th>Adequacy criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary values</td>
<td>Test selection:</td>
</tr>
<tr>
<td></td>
<td>• One dimensional</td>
</tr>
<tr>
<td></td>
<td>• Multidimensional</td>
</tr>
<tr>
<td>State transitions</td>
<td>Test selection:</td>
</tr>
<tr>
<td></td>
<td>• Random</td>
</tr>
<tr>
<td></td>
<td>• State coverage</td>
</tr>
<tr>
<td></td>
<td>• Transition coverage</td>
</tr>
<tr>
<td></td>
<td>• 0-switch coverage</td>
</tr>
<tr>
<td></td>
<td>• 1-switch coverage</td>
</tr>
<tr>
<td></td>
<td>• 2-switch coverage</td>
</tr>
</tbody>
</table>

(Continued)
code coverage is a term to describe which application code is exercised when the application is running, while test selection coverage relates to the percentage of the designed test cases with regard to the requirements in the selection criterion.

Please keep in mind that neither test selection nor test data adequacy criteria tell us anything about the quality of the software, only about the expected quality of the test cases. When deriving the test cases, be aware that one test case may exercise more than one test condition, and thus, there is the opportunity to optimise test case selection by combining multiple test coverage items in a single test case. At the end of this chapter, you can find a simple but complete example which explains these notions.

There is another important notion related to test selection. In a multivariate domain, we can apply different fault models to predict the consequences of faults.
The **single fault assumption** relies on the statistic that failures are only rarely the product of two or more simultaneous faults. Here we assume that fault in a program occurs due to the value of just one variable. On the contrary, the **multiple fault assumption** means that more than one component leads to the cause of the problem. Here we assume that fault in the program occurs due to the values of more than one variable.

Single fault assumption also means that we test only one input domain with one test case, and if it fails, then we know the location of the fault. By applying multiple fault assumption we can design fewer test cases; however, in the case of a failure, we should make additional test cases for bug localisation. There is no rule for which method is better.

4. The next step is to **establish the test data**. Test data are used to execute the tests, and can be generated by testers or by any appropriate automation tool (produced systematically or by using randomisation models). Test data may be recorded for reuse (e.g. in automated regression testing) or may be thrown away after usage (e.g. in error guessing).

    The time, cost and effectiveness of producing adequate test data are extremely important. Some data may be used for positive, others for negative testing. Typically, test data are created together with the test case they are intended to be used for. Test data can be generated manually, by copying from production or from legacy sources into the test environment, or by using automated test data generation tools. Test data creation takes many pre-steps or test environment configurations, which is very time-consuming. Note that concrete test cases may take a long time to create and may require a lot of maintenance.

5. Now we can **finalise the test case design**. A test case template contains a test case ID, a trace mapped to the respective test condition, test case name, test case description, precondition, postcondition, dependencies, test data, test steps, environment description, expected result, actual result, priority, status, expected average running time, comments and so on.

    For some software, it may be difficult to compute the proper outcome of a test case. In these cases, test oracles (sources of information for determining whether a test has passed or failed) are useful. Test management tools are a great help to manage the test cases.

    Where appropriate, the test cases should be recorded in the test case specification document. In this case, the traceability between the test basis, feature sets, test conditions, test coverage items and test cases should be explicitly described. It is advisable that the content of the test case specification document should be approved by the stakeholders.

    A **test suite** (or test set) is a collection of test cases or test procedures that are used to test software in order to show that it fulfils the specified set of behaviours. It contains detailed instructions for each set of test cases and information on the system configuration. Test suites are executed in specific test cycles. In this book we use a simplified notation for a test case, for example $TC = ([1, a, TRUE]; 99)$, where the test case TC has three input parameters 1, a, and TRUE, and expected value of 99. Sometimes, when the expected outcome is irrelevant with regard to a given example, we omit it from the test case. The notation $TS = \{([1, a, TRUE];$
Sometimes we simply list the steps to perform in a test case. Unfortunately, the specification sometimes contains issues that have to be corrected. For example, it may contain solutions or implementation details. This is completely wrong. The customer should not tell you how to implement the functionality they need. It is the developer’s task to implement the user’s needs in the best way. The test design is similar, as designed test cases have to test the implementation-independent specification, therefore cannot contain any implementation-dependent element.

Example for a test containing implementation details:

- Click on the LOGIN button
- Write ‘smith’ into the ‘Login name’ box
- Write ‘aw12K@S’ into the ‘Password’ box
- Click on ‘OK’

There are many ways you could activate logging in and submitting a log-in request. Think mouse clicks, taps on screen, keyboard shortcuts, voice control and so on.

Here is the implementation-independent version:

Example for implementation-independent tests – CORRECT:

- Login as ‘smith’
- Password is ‘aw12K@S’
- Action: confirm

When the implementation is ready, and the tests are executed, coverage analysis reports the untested parts of the code.

Now it is high time to create additional test cases to cover the uncovered code. However, it is not test design; it is test generation.

6. The next step is to design the test environment. The test environment consists of items that support test execution: software, hardware and network configuration. The test environment design is based on entities like test data, network, storage, servers and middleware. The test environment management has organisational and procedural aspects: designing, building, provisioning and cleaning up test environments requires well-established organisation.

There is an emerging technology concentrating on unifying testing and operation, called TestOps. It can be seen as a ‘process of using a combination of engineering artefacts, test artefacts, and field artefacts and applying the process of software analytics to improve the V&V [verification and validation] strategy’ (Kurani, 2018). In other words, the testing and operation processes are integrated into a single entity aimed at producing the best software system as quickly and efficiently as possible. TestOps provides continuous feedback to the testing team to ensure that failed tests are identified and addressed as early as possible. TestOps allows organisations to deliver high-quality software quickly and cheaply.
We do not consider test environment design in this book. This is a different, but emerging, topic.

7. Finally, validate all the important artefacts produced during the test design including the control of the existence of the bi-directional traceability between the test basis, test conditions, test cases and procedures.

The following list summarises the important factors to be considered during test design:

- test approach;
- level of risks;
- test design technique;
- test selection criteria, and their coverage;
- fault assumption;
- type of test defensiveness;
- whether it is concrete or abstract.

The output of the design process aggregates descriptions of prioritised test scenarios, test cases, appropriate test data and the controlled test environment.

Subsequent activities – test implementation and execution, test closure

During the implementation and execution phase, the designed test cases are implemented and executed. After implementing the test cases the test scripts are developed. A test script (or test procedure specification) is a document specifying a sequence of actions and data needed to carry out a test. A script typically has steps that describe how to use the application (which items to select from a menu, which buttons to press, and in which order) to perform an action on the test object. In some sense, a test script is an extended test case with implementation details.

When test automation is determined to be a useful option, this stage also contains implementing the automation scripts. Test execution automation is useful when the project is long term and repeated regression testing provides a positive cost–benefit.

The next step is to set up the test environment. In greenfield projects (which lack constraints imposed by prior work), extra time should be allowed for experiencing and learning. Even in a stable environment, organising communication channels and communicating security issues and software evolutions are challenging and require time.

In some cases (performance testing, security testing, maintenance testing, etc.), the testing environment should be a clone of the production environment. Moreover, it is not enough to set up the environment; a plan should be in place for an update and
reset when needed. In large-scale projects, several environments are present and need careful management and maintenance. Well-designed configuration management of the environment and testware is also important.

Next, the finalisation of the approach comes via test case implementation. At this point, everything is prepared for starting the execution, manually and/or automatically, which can be checked with the entry criteria. Some documents may help to localise the items that will be tested (test item transmittal documents or release notes).

The existence of the continuous integration environment is important for test execution (depending on the SDLC, and the type of project and product). The execution process is iterative, and in many cases, it is the longest step of the fundamental test process. Here, scripted techniques are often mixed with other dynamic techniques such as exploratory testing. The outcome of a test can usually be:

- **Passed**, if its actual result matches its expected result.
- **Failed**, if its actual result does not match its expected result.
- **Blocked**, if the test case cannot run because the preconditions for its execution are not fulfilled.
- **Incomplete**, if the test case did not complete its execution for various reasons.
- **Inconclusive**, if it produces a result that is not clear and requires further investigation and so on.

Once an issue is detected, it should be investigated whether it was really a result of a software defect. Logging the results is critical. Test logs are like gold dust and should be analysed thoroughly. The levels of logging depend on the test approach, the life cycle, regulatory rules, the project’s phase and so on. Logging information is especially useful in experience-based testing. In the case of failed tests, an incident management process begins. The analysis of the test results is extremely important since it serves as the basis of the feedback mechanisms. Testers have to gather all kinds of information during actual testing, which is then extracted and used to evaluate progress, data adequacy, coverage values, improvement possibilities and so on. For example, if the test design was not adequate, then some new tests must be designed. On the other hand, if code coverage was not appropriate, then new tests must be generated.

In extreme cases, the test management has to replan the test project to lower the quality risk. At this point there is important feedback for test design. In test evaluation and reporting, the test logs are checked against the clearly defined exit criteria (e.g. checking the test execution progress for high priority test cases, checking the status of all outstanding defects, etc.). Sometimes the exit criteria have to be modified with agreement from stakeholders. The result of the exit criteria evaluation is important information for all kinds of stakeholders; they want to know what happened during testing so that they can make the right decisions about the software project. A good test reporting tool is indispensable.

The **test closure** is a complete test report that gives a summary of the test project. It formally closes the project, collates all the test results, provides a detailed analysis, presents metrics to clients and assesses the risks with respect to the software as a whole.
WHY SHOULD TESTS BE DESIGNED?

We are not living in an ideal world. Nothing is perfect, including software. Developers make mistakes. The role of testers is to create tests, which reveal any mistakes made by the developers. The most secure way of doing this is to test the software for every possible input. However, this is impossible. Even for very simple code containing only two 32-bit integers, the number of test cases is $2^{64}$. Assuming each test execution takes 1 microsecond, the total testing time would be $2^{58}$ seconds, which is more than $10^{36}$ years and is incomparably more than the time elapsed from the Big Bang till now.

Well, even if exhaustive testing is not possible, there may be a general method or algorithm that generates reliable tests for every code. A reliable test set $T$ has the property that either for each $t \in T$, $t$ passes if and only if the code is correct or there is at least one test $t$ for which the software fails.

Unfortunately, it has been proved by Howden (1976) that there is no such method or algorithm. The consequence is that testing is not able to detect each bug. The only thing we can expect is to be able to design our test cases so that:

- we can find as many bugs as we can, and
- spend as little time with testing and bug fixing as possible.

Software testing is a team responsibility. High-quality and efficient test cases can only be achieved when everyone involved in test design is aware of this.

Another alternative could be to apply ad hoc methods to tests without designing any tests. Companies who apply this trial-and-error approach, suffer from a buggy code base and low-quality products. Ad hoc testing is similar to ad hoc coding – the result can be catastrophic. By applying test design, we are able to:

- create efficient and reliable test cases;
- stop creating new test cases when the available ones are enough;
- automate test cases when needed;
- document the design as part of the testing activity;
- make test design documents alternative specifications understandable by stakeholders;
- maintain the test cases in accordance to some modifications, that is delete obsolete and design new test cases;
- select test cases to be re-executed after modifications.

Of course, test design is necessary but not sufficient for some of the points above. It is important to note that applying test design does not bound thinking and creativity. On the contrary, it offers a strong ground for test case quality and for creative thinking.
Designing tests has some advantages:

- possibility of reproducing a test;
- increase of faults found;
- long-term maintenance of the system and test automation support;
- an objective approach to the testing process.

However, it is not always necessary to apply test design. It is only to be applied if it adds value. Sometimes it is unimportant to systematically test a component implementing a less important 'nice to have' feature. For important functions, however, more care is needed. The test quality for these components is provided by an adequate test design.

In many organisations, the test base is often converted directly into a test case. But the tester will still implicitly use test design techniques during the conversion. Basic knowledge of the test design techniques and their principles still has added value.

A standardised way of working makes the tests more independent, separating the design from the execution. Note that test selection criteria enable tests to be designed more efficiently.

Test design and test execution can be automated. Test suites can automatically be generated by software using model-based testing, model checking or symbolic execution. Model-based testing tools generate test cases from a model, such as an activity diagram, Petri net, finite-state machine (see Chapter 6), timed automaton, Gherkin code (see Chapter 12) and so on. Model checking is a technique for automatically verifying the correctness of a program or automaton. It ensures, for example, that all the paths of a simple program are exercised. However, it requires that both the model of the system and the specification are formulated in a precise mathematical language (e.g. B, Z or Larch). By symbolic execution, we are able to generate tests directly from the code that will expose the bug when the software is running. These methods do not make the test design superfluous. On the contrary, these techniques necessitate a good test model, which can only be created if appropriate test design techniques are applied.

**WHEN DO WE DESIGN TESTS?**

Nowadays it is widely accepted that test execution has to be done as early as possible. Especially, 'shift left' testing (where the testing processes move to the left on the project timeline) requires integration testing earlier. Test design has to be done as early as possible as well, but before coding in all cases. Specifications, use cases and user stories are appropriate source documents for implementing the code. In addition, designed test cases can significantly improve the documents the developer will start with. Good test cases are in fact examples that lead the developer during implementation.

Early test design is a very effective defect prevention method. Lots of bugs are design errors. Test design before coding involves the validation of the specification, removing most of the design problems.
Early test design eliminates the problem of involving testers too late and lowers the problem of increased testing effort. Moreover, in most cases, the overall testing effort can be reduced.

**IMPORTANT TEST DESIGN-RELATED CONSIDERATIONS**

Test design techniques are like lighting up the landing strip. They help you to find the right way in order to pick the right data. In safety-critical systems test design techniques are recommended by standards (e.g. DO 178C in aerospace systems, BS EN 50128 in the railway industry).

This section deals with the effect of test data selection and mutation testing on the test design. Test data is not random – it must be chosen **systematically**. Regarding test data selection we will use two important hypotheses in this book. These are explained in the following two subsections.

**Competent programmer hypothesis**

The first hypothesis is the **competent programmer hypothesis (CPH)**, which was identified by DeMillo et al. (1978), who observed that ‘Programmers have one great advantage that is almost never exploited: they create programs that are close to being correct.’ Developers do not implement software randomly. They start from a specification, and the software will be very similar to their expectations, hence, close to the specification.

CPH makes test design possible. If the implemented code could be anything then we should test the application to differentiate it from an infinite number of alternatives. This would need an infinite number of test cases. Fortunately, based on CPH we only have to test the application to separate it from the alternative specifications that are very close to the one being implemented.

To demonstrate CPH, let’s consider our **example for test selection criteria** in the ‘Present activity – test design’ section above, where the requirements are the following:

- **R1.** Input two integers, say x and y from the standard input device
- **R2.** If x and y are both negative, then print x + y to the standard output device
- **R3.** If x and y are both positive, then print their greatest common divisor to the standard output device
- **R4.** If x and y have opposite signs (one of them is positive and the other is negative), then print their product to the standard output device
- **R5.** If x × y is zero then print zero to the standard output device.

The test set T2 = {([-2, -3]; -5), ([4, 2]; 2), ([2, -4]; -8), ([2, 0]; 0) } is adequate. However, an alternative specification can be the following:

- R1. Input two integers, say x and y from the standard input device
- R2. If x and y are both negative, then print x + y to the standard output device
R3. If x and y are both positive and less than 100,000, then print their greatest common divisor to the standard output device, otherwise print the smaller value to the standard output device

R4. If x and y have opposite signs (one of them is positive and the other is negative), then print their product to the standard output device

R5. If x·y is zero then print zero to the standard output device.

Assume that the developer implemented this specification. In this case, T2 would not be adequate as all test cases would pass, yet the code is wrong. What’s more, any positive pair of test input below 100,000 would not be reliable. Assuming that the implementation can be anything we easily modify the specification again so that selecting any number of positive test input pairs would not be reliable.

Fortunately, because of the competent programmer hypothesis, it is unrealistic to implement the modified R3, since this requirement is far from the original, and the probability that a developer swaps them is minimal.

**Coupling effect**

The second hypothesis we will use in this book is the coupling effect, which states that a test data set that detects all simple faults in a program will also detect more complex faults. Here a simple fault means a fault that can be fixed by making a single change to a source statement. On the other hand, a complex fault is a fault that cannot be fixed by making a single change to a source statement. To reveal these complex faults, we suggest applying the Delta Debugging methodology, which automates debugging by systematically narrowing down the places where failures can occur (Zeller, 1999).

**Test design defensiveness**

The last notion that influences the test data selection and test design is its type of defensiveness. A defensive test design considers all possible input for designing the tests. Defensive testing designs test cases both for normal and abnormal preconditions. For instance, an invoked system should always test that the input file exists or not, is empty or not, contains the right data in the right order or not and so on.

**Contract-based test design** assumes that the preconditions stated in some ‘contract’ always hold and guarantee the fulfilments of some postconditions. In the previous example, we do not have to test the appropriateness of the input each time the system is called. Based on the contract, we assume that everything is OK with the input (i.e. preconditions are fulfilled), however, if security is an issue, then the defensive testing (and design) remains the only possibility.

**Black and white-box mutation testing**

Another important aspect of test design is mutation testing. In mutation testing, we are not directly concerned with testing the program. Rather we introduce faults into the code to see the reliability of our test design. Therefore, mutation testing is actually not testing, but ‘testing the tests’. The method is to slightly modify the original code, creating a mutant of it. We can make several mutants and a reliable test data set has
to ‘kill’ all of them. A test kills a mutant if the original code and the mutant behave differently. For example, if the code is \(y = x^2\) and the mutant is \(y = 2x\), then a test case \(x = 2\) will not kill the mutant while \(x = 3\) will.

The main problem with mutation testing is with cases of equivalent mutants. They are the false positives, that is equivalent mutants keep the program’s semantics unchanged, thus, cannot be detected by any test. Unfortunately, the decision whether two pieces of code are equivalent or not is an undecidable problem in general.

Mutation testing was introduced in the late 1970s by DeMillo et al. (1978) and Hamlet (1977). An excellent survey containing 250+ references on this topic is published by Jia and Harman (2011).

Mutation testing is applicable when the code is ready. The question is how to test our test cases when we have no code and thus white-box methods are out of the question.

Fortunately, a black-box solution can be introduced as well. Let’s assume that we have two specifications that are slightly different. The question is whether we can have a unique test set, which would test both specifications reliably. The answer is obviously no. To see why, let the task be implementing Specification A. Let Specification B be another specification, which is not equivalent to Specification A. Assume, on the contrary, that we have a unique perfect test set for both. Assume that the developer accidentally implemented Specification B bug free. In this case, no test will fail, though the code is not equivalent with Specification A, hence, some behaviour of A is not tested, which is a contradiction.

We can therefore slightly modify our specification and check whether there are any test cases that would be different for the test sets in the two versions. If there are, then the different test case(s) separate(s) the modified specification from the original one. Hence, we can create alternative specifications to prove that every test case we designed is necessary and there are no missing test cases. On the other hand, if an alternative specification consists of the same test cases, then some test is missing. Unfortunately, this process is very costly, and thus not always appropriate in practice.

**SUMMARY**

In this chapter we gave a general overview of software testing. We discussed its importance and the role of test design in the full testing life cycle. Among other notions, we explained test selection and test adequacy criteria, test coverage, test cases, a test suite and some related concepts, including the competent programmer hypothesis, coupling effect, test design defensiveness and black-box mutation testing.
abbreviations xvii
abstraction levels 217–18, 224–5
abstract test cases 7
acceptance criteria (AC) 172, 173–4, 234, 236–8, 242–3
acceptance tests 173–4
activity diagrams 229
ad hoc testing 17
AETG (Automatic Efficient Test Generator) System 204
all-actions criterion 99
all configurations criterion 100
all-events criterion 99
all loop-free paths criterion 100
all-n-transitions criterion 100
all-one-loop paths criterion 100
all-path criterion 100
all-round-trip paths criterion 100
all-states criterion 99
all-transition-pairs criterion 99
all-transitions criterion 99
all-transition-state criterion 101–2, 104–5, 112, 120, 226–7
all-transition–transition criterion 99, 102–3, 105
American Standard Code for Information Interchange (ASCII) 74
analysis 3, 4–5
AND 145, 147, 238–40
Arnicane, V. 88
authorisation 60–3, 75–8
Automatic Efficient Test Generator (AETG) System 204
Bach, J. 206, 209, 213
Balcer, M.J. 204, 253
base choice 180–1, 185–6, 193–4
behaviour-driven development (BDD) 177
Beizer, B. 2, 63
black-box testing xviii, 20–1, 78
boundaries 82
boundary value analysis (BVA)
definition xviii
domain-based testing 57, 63–71
equivalence partitioning (EP) and 71–8
extending use case testing 167–71
model-based testing (MBT) 223
state transition testing (STT) and 115–16
ticket vending machine (TVM) 273–5
business artefacts 156
Business Process Modelling Notation (BPMN) 222
business rule-based testing
cause–effect graph testing 144–53
decision table testing 129–44
definition xviii
key takeaways/exercises 153–4
buying process
combinative and combinatorial testing 200–2
defect prevention (DP) 53
domain-based testing 82–5
Gherkin-based MBT 247–9
model 288
risk analysis 40–1, 42
TVM specification 22–3, 24–5, 26
use case testing 164–71
BVA
see boundary value analysis (BVA)
CA Agile Requirements Designer 229–30
Carroll, J.M. 178
categories 234, 235–6, 241–2
category-partition method 204, 234, 253–4
cause–effect graphs
definition xviii
importance of 144
key takeaways/exercises 153
method evaluation 152–3
model-based testing (MBT) 219
RedShoe example 150–2
theoretical background 153
use in test design 145–50
carters 208, 210
checklist-based reviews 46
choices 234, 238
Chow, T.S. 99–100, 103, 104, 128
classification tree (CT) testing xviii, 198–200
closed boundaries 66, 67–8
Cockburn, A. 178
code coverage 11–12
Cohen, D.M. 204
Cohen, E.I. 79, 88
Cohn, M. 262
collapsed decision tables 131
Collatz conjecture 105–7
combinative testing
overview xviii, 179–81
diff-pair-N testing 182–5
diff-pair testing 181–2
method evaluation 202–3
optimisation and 35
ticket vending machine (TVM) 282–3
combinatorial testing
overview xviii, 179–81
base choice 185–6
classification tree (CT) 198–200
comparison of techniques 191–8
covering arrays 190–1
key takeaways/exercises 205
method evaluation 202–3
N-way testing 186–8
optimisation and 35
orthogonal array (OA) 188–91
theoretical background 203–5
ticket vending machine (TVM) 200–2
competent programmer hypothesis (CPH) xviii, 19–20
complexity
cost optimisation 35, 37–8
dividing and conquering 46
of test design xxii, xxv
computation errors 191–5
concrete test cases 7
concurrent systems 110–11
configuration testing 179, 180–1, 186–7
Conformiq Creator 229
continuous integration environment 16
contract-based test design 20
control 3, 4
control-flow errors 195–8
costs 34–7, 44
coupling effect 20
coverage domains 9–12
covering arrays 190–1
CPH (competent programmer hypothesis) xviii, 19–20
DAG (directed acyclic graph) 144
data faults 68–71
Davis, J.C. 143
decision table testing
constructing decision tables 135–6
definition xviii
importance of 153–4
method evaluation 142–3
RedShoe example 130–5, 139–42
theoretical background 143–4
ticket vending machine (TVM) 136–9, 283
defect correction 34–7
defect prevention (DP)
overview xviii, 44–6, 54
cycle 45
methods 46–7
ticket vending machine (TVM) 51–3
‘two lifts in a 10-storey building’ example 47–51
defensive test design 20
definition of done 172
delete selection 23, 41–2, 53
Delphi technique 31
DeMilo, R.A. 21
dependence diagrams 140
deterministic finite automaton (DFA) 93
diff-pair-N testing 182–5
diff-pair testing xviii, 181–2, 194–5, 196–8, 200–1
directed acyclic graph (DAG) 144
divide-and-conquer approach 46, 47, 219
domain analysis 58, 78–81
domain-based testing
overview xviii, 57–8
boundary value analysis 63–71
challenges in 82
domain analysis 78–81
equivalence partitioning (EP) 58–63
key takeaways/exercises 88–9
method evaluation 86–7
theoretical background 87–8
ticket vending machine (TVM) 82–5
domain boundaries 57–8
domain errors 88
DP
see defect prevention (DP)
EFSM (extended finite-state machine) 93, 128, 222, 256, 267
elevators 47–51
Elmendorf, B. 144, 153
EP
see equivalence partitioning (EP)
epics 172
equivalence classes 58
equivalence partitioning (EP)
boundary value analysis (BVA) and 65–8, 71–8
definition xix
domain-based testing 57, 58–63
examples 60–3
extending use case testing 167–71
model-based testing (MBT) 223
as partition testing 87–8
state transition testing (STT) and 115
ticket vending machine (TVM) 273–5
error handling
defect prevention (DP) 53
risk analysis 41–2
TVM specification 23
ET
see exploratory testing (ET)
evaluation of risk 30–2
examples
configuration testing 180–1, 186–7
equivalence partitioning (EP) 60–3
implementation-independent tests 14
login authorisation/ functionality 5, 14, 60–3, 75–8
Object Constraint Language (OCL) 224
price reduction for online book purchasing example 234–8
RedShoe example 130–5, 139–42, 150–2, 191–8
RoboDog 93–6, 103–5
sequence covering arrays 204–5
square-filling xxii–xxvi
taxi driver’s decision 79–81
test cases 8
test objects and test items 5
test selection criteria (TSC) 9–11, 19–20

definition xviii
327
two lifts in a 10-storey building' example 47–51, 107–18
see also ticket vending machine (TVM)
exclusive constraints 148
exercises
cause–effect graphs 153–4
combinatorial testing 205
domain-based testing 89
on-the-fly test design 214
Gherkin-based MBT 255
GraphWalker 268
model-based testing (MBT) 232
scenario-based testing 178
solutions to 297–320
state transition testing (STT) 128
exit criteria 16
exploratory testing (ET)
definition xix, 213
on-the-fly test design 206–8
method evaluation 212–13
theoretical background 212–14
ticket vending machine (TVM) 211–12
extended-entry decision table 131, 134, 139–42
extended finite-state machine (EFSM) 93, 128, 222, 256, 267
fault models 58–9, 61–3
feature-driven development (FDD) 175
features 4, 172, 234–5, 241
final version 22–3
finite-state machine (FSM) 92–3, 127, 222
forks 110
FURPS+ model (Functionality, Usability, Reliability, Performance, Security) 172
Gerhart, S. 87
Gherkin-based MBT
overview 233–4
Harmony (MBT tool) 233–4
key takeaways/exercises 254–5
method evaluation 252–3
modelling language 238–44, 254
price reduction for online book purchasing example 234–8
syntax 174–5, 234
test design maintenance 244–5
theoretical background 253–4
ticket vending machine (TVM) 245–52, 283
GIVEN 238–40
glossary xviii–xx
GO/GOES 243–4
Goodenough, J. 87
graphical user interface (GUI) tests 262
GraphWalker (GW)
overview 268–9
key takeaways/exercises 268
making it work 257–9
method evaluation 267–8
pom.xml for 265, 294–6
test code for demonstrating 291–3
ticket vending machine (TVM) 259–67, 283
greedy algorithms 204
Grimm, K. 204
Grochtmann, M. 204
guard conditions 91
Gutjahr, W.J. 38
Hamlet, D. 87
Hamlet, R.G. 21
happy paths
see positive test cases
Harel, D. 110, 127, 222
Harman, M. 21
Harmony (MBT tool) 252–4
heuristic search 204
hierarchically extended decision tables 139–42
high-level test cases 7
high-risk functions (HRF) 36
Hopcroft, J. 93
Howden, W.E. 17
identity 145, 147
implementation-dependent tests 14, 65
implementation-independent tests 14, 47
inactivity
defect prevention (DP) 53
risk analysis 40, 42
TVM specification 23
inclusion constraints 148
initial screen 22, 40, 52
IN points 63, 66, 68, 71–2, 74, 80–1
input domains 57
International Software Testing Qualification Board (ISTQB) 2
invalid partitions 71–2
invalid transitions 92
INVEST (independent, negotiable, valuable, estimable, small, testable) 172
IS 238–40
ISTQB (International Software Testing Qualification Board) 2
Itkonen, J. 212
Java 257–9, 266–7
Java Development Kit (JDK) 257
Jeffries, R. 178
Jeng, B. 88
Jia, Y. 21
Johnston, F.J.J. 143
joins 110
Kaner, C. 155, 206, 213
KDT (keyword-driven testing) 225–6
key takeaways
cause–effect graphs 153
combinatorial testing 205
defect prevention (DP) 54
domain-based testing 88–9
on-the-fly testing 214
Gherkin-based MBT 254–5
GraphWalker 268
model-based testing (MBT) 232
risk management 43
scenario-based testing 178
state transition testing (STT) 128
keyword-driven testing (KDT) 225–6
Kuhn, R. 104, 179–80, 204
labelled transition systems 127
lifts 47–51, 107–18
limited-entry decision tables 130–1, 132–3
linearly ordered partitions 73
logging information 16
logical operators 145–7
login authorisation/functionality 5, 14, 60–3, 75–8
low-risk function (LRF) 36

Mandl, R. 204
Mask constraints 149
Mathur, P.A. 153
Maven 257–9

MBT
see model-based testing (MBT)

Mealy machines 93, 127

method evaluation
cause–effect graphs 152–3
combinative testing 202–3
combinatorial testing 202–3
decision table testing 142–3
domain-based testing 86–7
on-the-fly testing 212–13
GraphWalker 267–8
Harmony (MBT tool) 252–3
model-based testing (MBT) 230–1
scenario-based testing 175–7
state transition testing (STT) 126–7

model-based testing (MBT)
overview 217
abstraction levels in 224–5
CA Agile Requirements Designer 229–30
Conformiq Creator 229
defect prevention (DP) 46
definition xix, 218–19
Gherkin-based MBT 233–55
GraphWalker 256–68
key takeaways/exercises 232
keyword-driven testing (KDT) 225–6
method evaluation 230–1
modelling languages 222–3
process of 219–21
test case generation 228–9
test design and 17, 219, 223–4
test selection criteria (TSC) 226–8

Tricentis Tosca TestSuite 229
model checking 17
modelling 219–20
modelling languages E(FSM) 256

Gherkin 233, 234, 238–44, 254
GraphWalker 256
model-based testing (MBT) 222–3
Unified Modelling Language (UML) 222, 223
models 217–18, 223–4
monitoring 3, 4
Moore machines 93, 127
multi-layer structure 241
multiple fault assumptions 13, 59, 63
multiple techniques 105, 107–18
mutation testing xix, 20–1, 78
Myers, G.J. 149, 150–1, 153

NAND 146, 147
negative test cases 7, 37
non-deterministic finite automata 93
NOR 146, 147
NOT 145, 147
null 59, 73
null transitions 92
N-way testing 186–8

OA (orthogonal array) xix, 188–91
Object Constraint Language (OCL) 223–4
OFF points 63, 65–8, 71–5, 80
online shop example 195–8
only constraints 149
ON points 63, 65–8, 71–5, 79–81
on-the-fly test design
overview xix, 206
exploratory testing (ET) 206–8
key takeaways/exercises 214
method evaluation 212–13
session-based testing 208–10
theoretical background 213–14
ticket vending machine (TVM) 211–12

open boundaries 65
operator (predicate) faults 64–8, 78–9
optimisation
risk and 38–9
software testing 37–8
state transition testing (STT) 100–1

OR 146, 147
ordering 64
orthogonal array (OA) xix, 188–91
Ostrand, T.J. 87, 204, 253
OUT points 63, 65, 67–8, 71–4, 79–81

‘packaging and delivery service’ example 191–5
partition testing 87–8
passwords 60–3
payment process
see buying process
planning 3–4

pom.xml for GraphWalker 265, 294–6
positive test cases 7, 37
predicate faults 64–8, 78–9
price reduction for online book purchasing example 234–8
prime paths 100
PROOF method (Past, Results, Obstacles, Outlook, Feelings) 209–10
random walking 100–1, 263
Rautiainen, K. 212

RedShoe example
cause–effect graphs 150–2
decision table testing 130–5, 139–42
testing computation errors 191–5
testing control-flow errors 195–8
reduced mode 23, 25–6, 53, 273–5, 278, 282
Reid, S.C. 88
replication 243–4
Require constraints 148–9
requirements’ feasibility 47–51
resource-based prioritisation 38
reviews 5, 46

risk
analysis xix, 5, 30, 39–42
costs, quality and 8
definition 29–30
optimisation and 38–9
scoring 30–2
ticket vending machine (TVM) 39–42

risk-based prioritisation 38–9
risk management
key takeaways 43
process of 30, 32–4
purpose of 34–7
risk poker 30–1
RoboDog example 93–6, 103–5
round-trip paths 100
SBE (specification by example) 8, 46–7, 51–3
SBTM (session-based test management) 208–10
scenario-based testing overview xix, 155–6
key takeaways/exercises 178
method evaluation 175–7
theoretical background 178
use cases 156–64
use case TVM example 164–71
user stories 171–5
script-oriented test design 206
SDL (Specification and Description Language) 127
SDLC (software development life cycle) 3, 178
selecting tickets see ticket selection
sequence covering arrays 204–5
service/API testing 262
session-based testing xix, 208–10, 213–14
simple paths 100
single fault assumptions 13, 59, 62–3
software development life cycle (SDLC) 3, 178
software testing costs 34–7
importance of 1–2
optimising 37–8
special characters 74
Specification and Description Language (SDL) 127
specification by example (SBE) 8, 46–7, 51–3
specifications from requirements 47–51
ticket vending machine (TVM) 22–6, 51–3
spikes 172
square-filling example xxii–xxvi
Srivastava, P. R. 153
standard reviews 46
statecharts 110, 127, 222, 271–3
stateful systems 90–1, 128
stateless systems 90
state machines 127
statement testing 87–8
state transition diagrams/graphs 91–3, 97–8, 128, 175, 219, 243–4
state transition tables 91–2
state transition testing (STT)
Collatz conjecture 105–7
definition xix
key takeaways/exercises 128
method evaluation 126–7
model-based testing (MBT) 223
RoboDog example 93–6, 103–5
stateful and stateless systems 90
states, transitions, conditions 91–3
test selection criteria (TSC) 98–107
theoretical background 127–8
ticket vending machine (TVM) 118–26
validation 97–8
strategy 3
structure diagrams 229
STT see state transition testing (STT)
sub-acceptance criteria 242–3
successful transactions defect prevention (DP) 53
Gherkin-based MBT 246, 248–9
TVM specification 23, 25
TAC see test data adequacy criteria (TAC)
taxi driver’s decision example 79–81
Taylor, R. 87
TCP (test case prioritisation techniques) 38
TDD (test-driven development) 221, 262
test adaptation 221
test analysis 3, 4–5
test approaches 3–4, 6
test artefacts 156
test automation 15, 17
test automation pyramids 262–3
test bases 4, 6–7
test case prioritisation techniques (TCP) 38
test cases definition xix, 2
designing 13
eamples 8
maintenance 244–5
model-based testing (MBT) 220–1, 228–9
prioritisation 38
ticket vending machine (TVM) 276–82
types 7–8
test case selection criteria see test selection criteria (TSC)
test closure 3, 16
test conditions 4–5
test control 3, 4
test data 13, 19–20
test data adequacy criteria (TAC) xix, 8–9, 10–11
test design overview 285–7
defensiveness 20
definition xix, 2
important considerations 19–20
maintenance 244–5
output 287
preceding activities 2–5
purpose of 17–18
seven steps of 6–15
in software development life cycle (SDLC) 3
subsequent activities 15–16
techniques xxv, 6–7, 283–4, 286–7
timing of 18–19
using multiple techniques together 107–18
test-driven development (TDD) 221, 262
test environments 14–16
test-first approach 4, 233–4
test implementation and execution 3, 15–16, 221
test items 5
test monitoring 3, 4
test objects 4–5
TestOps 14
test paths xix
test planning 3–4
test procedure specifications
   see test scripts
test scenarios 4–5
test scripts xix, 15
test selection criteria (TSC)
   overview xx, xxv, 7–8
   boundary value analysis (BVA) 63, 72
   examples 9–11, 19–20
   model-based testing (MBT) 220, 226–8
   state transition testing (STT) 99–105, 120
test strategy 3
test suites xx, 13–14, 38–9
themes 172
THEN 238–40
theoretical background
   cause–effect graphs 153
   combinatorial testing 203–5
   decision table testing 143–4
   domain-based testing 87–8
   on-the-fly testing 213–14
   Gherkin-based MBT 253–4
   scenario-based testing 178
   state transition testing (STT) 127–8
   test data adequacy criteria (TAC) 9
thoroughness 8–9, 35, 103–5
ticket selection
   classification tree (CT) 199–200
   decision table testing 136–9
   defect prevention (DP) 52
   Gherkin-based MBT 245–7, 249–52
GraphWalker 259–67
   model 288–90
   state transition testing (STT) 118–26, 270–3
   TVM specification 22, 23
   ticket types 22, 52
   ticket vending machine (TVM)
      boundary value analysis (BVA) 273–5
      combinative testing 282–3
      combinatorial testing 200–2
      decision table testing 136–9, 283
      domain-based testing 82–5
      on-the-fly test design 211–12
      Gherkin-based MBT 245–52, 283
      GraphWalker 259–67, 283
      method comparison 269–84
      models 288–90
      risk analysis 39–42
      specification 22–6, 51–3
      state transition testing (STT) 118–26, 270–3
      test cases 276–82
      use case testing 282
      transaction OK 248–52, 288
      Tricentis Tosca TestSuite 229
      TSC
         see test selection criteria (TSC)
      ‘two lifts in a 10-storey building’ example 47–51, 107–18
      undefined transitions 92
      Unified Modelling Language (UML) 222, 223
      unit tests 262
      unlabelled transition systems 127
use cases
   overview 156
   enter/leave building zones example 157–64
   extended by EP and BVA 167–8
   method evaluation 175–7
   in test design xx, 156–7
   test selection criterion (TSC) 164
   theoretical background 178
   ticket vending machine (TVM) 164–71, 282
user interfaces (UI) 262
user stories
   overview 171–2
   definition xx
   method evaluation 175–7
   test design and 173–4
   test scenarios and 5
   theoretical background 178
   ticket vending machine (TVM) 174–5
valid partitions 71–2
valid transitions 92
variable (predicate) faults 64–8, 78–9
verification and validation (V&V) 5, 14, 15, 221
Wake, Bill 172
Weyuker, E. 87, 88
WHEN 238–40
White, L.J. 79, 88
white-box testing xx, 20–1, 87–8
yEd Graph Editor 256

zero 59, 73
PRACTICAL TEST DESIGN
Selection of traditional and automated test design techniques
István Forgács and Attila Kovács

Reliable test design is important in software testing; without it, defects in software may remain undetected, making the software poor quality and causing dissatisfaction among users.

This book presents the key test design techniques, in line with ISTQB, and explains when and how to use them, including in combination, with practical, real-life examples. Automated test design methods are also explored. Tips and exercises are included throughout the book, allowing you to test your knowledge as you progress.

- Understand the importance of well-planned test design
- Learn the necessary steps to undertake before test design
- Explore the use of traditional test design, including domain-based testing, state transition testing, business rule-based testing, scenario-based testing, combinatorial and combinatorial testing, and on-the-fly testing
- Examine automated test design techniques, with model-based testing, Gherkin, and GraphWalker

ABOUT THE AUTHORS
István Forgács is the founder and CEO of 4Test-Plus. He is a member of the Agile Extension Board of ISTQB and co-author of the book Agile Testing Foundations.

Attila Kovács is an associate professor at Eötvös Loránd University. He is an ISTQB and IREB trainer, as well as a member of the Hungarian Testing Board.

You might also be interested in:

This is the testing book I've been waiting for.
Elaine Weyuker, University Distinguished Professor; Member, US National Academy of Engineering; ACM Fellow, IEEE Fellow

A must-read for anyone who wants more understanding of test design techniques and especially using them together.
Kari Kakkonen, Testing Trainer and Consultant at Knowit; ISTQB Secretary, and co-author of Agile Testing Foundations

Finally, a book that makes the test design techniques available to everybody regardless of their level of experience.
Francisca Cano Ortiz, Head of Software Quality Assurance, The Stars Group

Cover photo: iStock © Oleg Lopatkin
Paperback available