

# AGILE TESTING FOUNDATIONS

An ISTQB Foundation Level  
Agile Tester guide

Rex Black (editor)



# AGILE TESTING FOUNDATIONS

## **BCS, THE CHARTERED INSTITUTE FOR IT**

BCS, The Chartered Institute for IT, champions the global IT profession and the interests of individuals engaged in that profession for the benefit of all. We promote wider social and economic progress through the advancement of information technology science and practice. We bring together industry, academics, practitioners and government to share knowledge, promote new thinking, inform the design of new curricula, shape public policy and inform the public.

Our vision is to be a world-class organisation for IT. Our 75,000-strong membership includes practitioners, businesses, academics and students in the UK and internationally. We deliver a range of professional development tools for practitioners and employees. A leading IT qualification body, we offer a range of widely recognised qualifications.

### **Further Information**

BCS, The Chartered Institute for IT,  
First Floor, Block D,  
North Star House, North Star Avenue,  
Swindon, SN2 1FA, UK.  
T +44 (0) 1793 417 424  
F +44 (0) 1793 417 444  
[www.bcs.org/contact](http://www.bcs.org/contact)

<http://shop.bcs.org/>



The  
Chartered  
Institute  
for IT

# **AGILE TESTING FOUNDATIONS**

An ISTQB Foundation level Agile  
tester guide

Edited by Rex Black, with contributions from  
Gerry Coleman, Marie Walsh, Bertrand Cornanger,  
Istvan Forgács, Kari Kakkonen and Jan Sabak



© BCS Learning & Development Ltd 2017

The right of Rex Black to be identified as author of this work has been asserted by him in accordance with sections 77 and 78 of the Copyright, Designs and Patents Act 1988.

All rights reserved. Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted by the Copyright Designs and Patents Act 1988, no part of this publication may be reproduced, stored or transmitted in any form or by any means, except with the prior permission in writing of the publisher, or in the case of reprographic reproduction, in accordance with the terms of the licences issued by the Copyright Licensing Agency. Enquiries for permission to reproduce material outside those terms should be directed to the publisher.

All trademarks, registered names etc. acknowledged in this publication are the property of their respective owners.

BCS and the BCS logo are the registered trademarks of the British Computer Society, charity number 292786 (BCS).

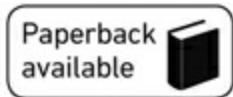
Published by BCS Learning & Development Ltd, a wholly owned subsidiary of BCS, The Chartered Institute for IT, First Floor, Block D, North Star House, North Star Avenue, Swindon, SN2 1FA, UK.  
[www.bcs.org](http://www.bcs.org)

Paperback ISBN: 978-1-78017-33-68

PDF ISBN-13: 978-1-78017-33-75

EPUB ISBN-13: 978-1-78017-33-82

Kindle ISBN-13: 978-1-78017-33-99



British Cataloguing in Publication Data.

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

#### Disclaimer:

The views expressed in this book are those of the authors and do not necessarily reflect the views of the Institute or BCS Learning & Development Ltd except where explicitly stated as such. Although every care has been taken by the authors and BCS Learning & Development Ltd in the preparation of the publication, no warranty is given by the authors or BCS Learning & Development Ltd as publisher as

to the accuracy or completeness of the information contained within it and neither the authors nor BCS Learning & Development Ltd shall be responsible or liable for any loss or damage whatsoever arising by virtue of such information or any instructions or advice contained within this publication or by any of the aforementioned.

Typeset by Lapiz Digital Services, Chennai, India.

# CONTENTS

[List of figures and tables](#)

[Authors](#)

[Foreword](#)

[Acknowledgements](#)

[Abbreviations](#)

[Glossary](#)

[Preface](#)

## **AGILE SOFTWARE DEVELOPMENT**

[1.1 The fundamentals of Agile software development](#)

[1.2 Aspects of Agile approaches](#)

[References](#)

[Further reading](#)

[Websites](#)

## **FUNDAMENTAL AGILE TESTING PRINCIPLES, PRACTICES AND PROCESSES**

[2.1 The differences between testing in traditional and Agile  
approaches](#)

[2.2 Status of testing in Agile projects](#)

[2.3 Role and skills of a tester in an Agile team](#)

[References](#)

[Further reading](#)

[Websites](#)

[Notes](#)

## **AGILE TESTING METHODS, TECHNIQUES AND TOOLS**

[3.1 Agile testing methods](#)

[3.2 Assessing quality risks and estimating test effort](#)

[3.3 Techniques in Agile projects](#)

[3.4 Tools in Agile projects](#)

[References](#)

[Further reading](#)

[Websites](#)

[Notes](#)

## **APPENDIX: TEST YOUR KNOWLEDGE ANSWERS**

### **AGILE TESTER SAMPLE EXAM**

[Questions](#)

[Answers](#)

[Index](#)

# LIST OF FIGURES AND TABLES

- Figure 1.1** Overview of the Agile development process
- Figure 1.2** The Agile Manifesto key values
- Figure 1.3** Twelve principles of the Agile Manifesto
- Figure 1.4** Transfer of responsibility from developer to tester and vice versa
- Figure 1.5** Potential communication channels in an Agile team
- Figure 1.6** Example of a CRC card for a university student
- Figure 1.7** Example of a Scrum process
- Figure 1.8** Example of a Kanban board
- Figure 1.9** Example of product backlog
- Figure 1.10** Stories focus on functionality rather than implementation
- Figure 1.11** Stories focus on user value
- Figure 1.12** Example of tests linked to user stories
- Figure 1.13** Example of a continuous integration system
- Figure 1.14** Release and Iteration planning
- Figure 2.1** Example of a story card
- Figure 2.2** Waterfall model
- Figure 2.3** V-model for software development
- Figure 2.4** Agile Scrum approach
- Figure 2.5** Agile development and test
- Figure 2.6** Features delivered per iteration

- Figure 2.7** Example of a task board  
**Figure 2.8** Example of a burndown chart  
**Figure 2.9** Example of a product burndown chart  
**Figure 2.10** Example of a Niko-niko calendar  
**Figure 2.11** Testing activities throughout iteration activities  
**Figure 3.1** Process of creating tests and code with TDD  
**Figure 3.2** The test pyramid with typical test levels  
**Figure 3.3** The test pyramid with exploratory cloud  
**Figure 3.4** Agile testing quadrants  
**Figure 3.5** An example of a simple test charter done with mind mapping tool XMind  
**Figure 3.6** Optimising failure cost  
**Figure 3.7** Failure costs differ for each story  
**Figure 3.8** Optimising failure costs with multiple test methods  
**Figure 3.9** Quality risk analysis template  
**Figure 3.10** Risk poker flowchart  
**Figure 3.11** Visualising user story risk  
**Figure 3.12** An example of user story risk  
**Figure 3.13** Estimating absolute area  
**Figure 3.14** Risk and planning poker flow  
**Figure 3.15** Example of equivalence partitioning and boundary value analysis  
**Figure 3.16** Example of a state transition diagram  
**Figure 3.17** Gym system architecture  
**Figure 3.18** An example of iteration backlog  
**Figure 3.19** An example of a task board  
**Figure 3.20** Example of a mind map  
**Figure 4.1** Equivalence partitions for the account test  
**Figure 5.1** State transitions for ATM PIN user story

- Table 1.1** Types of user stories  
**Table 1.2** User story example  
**Table 1.3** Topics in retrospectives  
**Table 3.1** Different Agile team practices for Scrum teams and testers  
**Table 3.2** Typical tasks of Sprint Zero and their implications  
**Table 3.3** Different testing-related pairs in an Agile team  
**Table 3.4** Risk-based test method selection  
**Table 3.5** Example of an acceptance test-driven development table  
**Table 3.6** Gherkin format tests  
**Table 4.1** Example of ATDD solution

## ABOUT THE AUTHORS

**Rex Black** has more than 30 years of software and systems engineering experience. He is President of RBCS ([www.rbcus-us.com](http://www.rbcus-us.com)), a leader in software, hardware and systems testing. For almost 25 years RBCS has delivered training, consulting and expert services for software, hardware and systems testing and quality. Employing the industry's most experienced and recognised consultants, RBCS builds and improves testing groups and provides testing experts for hundreds of clients worldwide. Ranging from Fortune 500 companies to start-ups, RBCS clients save time and money through higher quality, reduced risk of production failures, improved product development, improved reputation and more.

Rex has authored more than a dozen training courses, many of them ISTQB® accredited. These materials have been used to train tens of thousands of people internationally, both directly through RBCS on-site, virtual, public and e-learning courses and through a worldwide network of licenced training partners in Europe, Africa, the Middle East, Australasia, East Asia and Southern Asia. In addition to for-profit training courses, he has presented an hour-long, monthly, free webinar for more than seven years; this webinar series has reached more than 40,000 registrants. Rex has written numerous articles, presented hundreds of papers, workshops, webinars and seminars, and given more than 100 keynotes and other speeches at conferences and

events around the world.

Rex is a past president of the International Software Testing Qualifications Board (ISTQB®) and of the American Software Testing Qualifications Board (ASTQB). He remains active in both organisations. He leads the Agile Working Group, is the coordinator for the Advanced Test Manager and Expert Test Manager syllabus teams and participates as an author and reviewer in a number of other syllabus teams as well. He holds a BS degree in Computer Science and Engineering from the University of California, Los Angeles, and a Master's Certificate in Software Testing from Villanova University.

Rex is the most prolific author practising in the field of software testing today. His popular first book, *Managing the Testing Process*, has sold more than 100,000 copies around the world, including Japanese, Chinese and Indian releases, and is now in its third edition. His other books on testing, *Agile Testing Foundations*, *Advanced Software Testing: Volumes I, II, and III*, *Critical Testing Processes*, *Foundations of Software Testing*, *Pragmatic Software Testing*, *Fundamentos de Pruebas de Software*, *Testing Metrics*, *Improving the Testing Process*, *Improving the Software Process* and *The Expert Test Manager* have also sold tens of thousands of copies, including Spanish, Chinese, Japanese, Hebrew, Hungarian, Indian and Russian editions.

You can find Rex on Twitter ([@RBCS](#)), on Facebook ([@TestingImprovedbyRBCS](#)), on LinkedIn (rex-black) and on YouTube ([www.youtube.com/user/RBCSINC](http://www.youtube.com/user/RBCSINC)).

**Gerry Coleman** is a member of the Department of Visual and Human-Centred Computing at Dundalk Institute of Technology, Ireland, where he specialises in teaching software engineering, with a particular focus on Agile methodologies and software testing.

He has more than 30 years of experience in the IT industry as a practitioner,

academic and researcher and received his PhD from Dublin City University for research into software process improvement in the indigenous Irish software industry. Prior to taking up an academic position, he worked as a software developer, systems/business analyst and project manager in the financial sector and was a senior consultant, with responsibility for research and technology training and transfer, in the Centre for Software Engineering at Dublin City University. He now engages in software development consultancy, bringing the benefits of Scrum to small software teams.

Having published in excess of 100 papers, and presented at conferences and events worldwide, Gerry is a programme committee member for several widely recognised international conferences and has for many years been a reviewer for the *Journal of Systems and Software* and the *Journal of Software Maintenance and Evolution: Research and Practice*. He is co-author of the ISTQB® Foundation Level Agile Tester Extension syllabus and is a former director of the Irish Software Testing Board.

**Bertrand Cornanger** is an experienced software quality consultant with 23 years experience. For the last 10 years he has been a member of the French Software Testing Board, where he has served as secretary and treasurer.

After several years developing software engineering projects for customers and for internal purposes at NEC Computer International, Bertrand earned his MSc in Quality and Safety of Computer Systems. He has also studied Industrial Computer Systems, Networks and International Culture at ITII (Institut des Techniques de l'Ingénieur de l'Industrie) in France.

He started a new career in information systems as an outsourced test teams manager, organising manual and automated tests. The detection of numerous quality defects during acceptance tests, and shortcomings in the projects, led him to develop software quality consulting processes as a complement to software testing: namely, test strategies, software risk analysis, root cause analysis and test maturity audits. These were intended for use by large

organisations such as insurance companies to improve testing as well as development processes. At the same time, he developed training, coaching and change management processes in the areas of ISTQB® testing and Requirement Engineering Qualification Board (REQB) requirements to improve testers' skills and competences.

He was soon involved in the promotion of Agile, as Agile offers a good opportunity to implement efficient defect prevention and test practices. He has written a number of papers about software quality and Agile in France, has given several conferences on the topic around the world and is co-author of the ISTQB® Foundation Level Agile Tester Extension syllabus.

**Istvan Forgács** has a PhD in computer science and was originally a researcher at the Hungarian Academy of Sciences. He has published more than 25 scientific articles in leading journals and conference proceedings, and invented a partly automated method whose implementation efficiently solved the Year 2000 (Y2K) problem so that one tester was able to review 300,000 lines of code (LOC)/day.

He is the founder and CEO of 4Test-Plus. He led the team of 4D Soft for several European Union (EU) and national grants. His research interests include debugging, code comprehension, static and dynamic analysis, Agile testing and model-based testing. He is a member of the Agile Extension Board of ISTQB®.

Dr Forgács is the creator of the 4Test MBT method by which model-based testing becomes Agile, and easier to apply.

**Kari Kakkonen** has an MSc in Industrial Management, with Minor in Information Technology, from Aalto University ([www.aalto.fi](http://www.aalto.fi)). He has also studied at the University of Wisconsin-Madison ([www.wisc.edu](http://www.wisc.edu)) in the United States. He has the ISTQB® Expert Level Test Management Full, ISTQB® Agile Tester, Scrum Master and SAFe Agilist certificates, and

works mostly with Agile testing, Lean, test automation and DevOps. He has been working with testing tool implementations, software development and testing processes and projects, consulting and training since 1996. He has worked in various industries, including banking, telecom, embedded software, public sector and commerce.

Kari is currently working in Finland at Knowit ([www.knowit.fi](http://www.knowit.fi)), which is a Nordic IT services company well known for testing consultancy and software development, and other innovative IT related services. He is one of the most sought after Agile testing trainers and consultants in Finland and abroad, and has particular skills in getting his audience to understand the software testing concepts and inspiring people to strive for betterment in their work.

Kari is treasurer of ISTQB® ([www.istqb.org](http://www.istqb.org)), chairman of the Finnish Software Testing Board (FiSTB) ([www.fistb.fi](http://www.fistb.fi)), board member of the Finnish Association of Software Testers (<http://testausosy.fi>), auditor of the Robot Framework Foundation (<http://robotframework.org/foundation>), and a member of Agile Finland (<http://agile.fi>). He was included in *Tietoviikko* magazine's 100 most influential people in the IT 2010 listing.

Kari sings in the Drambuie Singing Group, paddles in his sea kayak, chairs the kayaking club Drumsö Kanotister and snowboards in the Finnish hills and the Alps. Kari is happily married with one son.

You can find Kari on Twitter (<https://twitter.com/kkakkonen>) and LinkedIn ([www.fi.linkedin.com/in/karikakkonen](http://www.fi.linkedin.com/in/karikakkonen)).

**Jan Sabak** is a software quality assurance expert. For 20 years he has been working in testing and on the quality of software and hardware. He holds a MSc in Computer Science from the Warsaw University of Technology. He built and managed quality assurance departments in Matrix.pl, IMPAQ and Infovide. Currently he works in his own consulting company, AmberTeam Testing, which strives to assure the peaceful sleep of chief information

officers (CIOs) and project managers through risk measurement and management. He is an active promoter of the knowledge and culture of the quality of software development, and is a president of the Revision Board of SJSI (Association for the Quality of Information Systems) also known as ISTQB's Polish Testing Board. He possesses the ISTQB® CTFL and CTAL-Full certificates, as well as the ISTQB® CTFL Agile Extension.

**Marie Walsh** is a freelance programme test manager, having started her IT career in 1998 after a career change from customer service management roles.

During her IT career, Marie has worked as a technical tester, test automation specialist, business analyst, data analyst, systems analyst, test manager, release and implementation manager, change manager, problem manager, project manager and solutions delivery manager. Her experience spans a large range of technologies, methodologies and disciplines across many industry sectors, including banking, finance, telecommunications, utilities, health and government.

Marie has been involved in various Agile development teams since 2002. Her first Agile project was with a middleware development team that had adopted an Extreme Programming (XP) Agile approach. Since 2002 she has worked with many other companies, who subscribed to various Agile practices including Microsoft's Solutions Framework, Scrum, Kanban and many variances of other Agile practices. Recent roles have involved problem management of struggling Agile development teams, and implementing techniques to assist the teams to stabilise and recover.

Marie's career focus is on the testing discipline, and her diverse roles have provided her with a solid understanding of the importance of testing within IT project lifecycles. Her passion is to improve the image of testing, and to work with teams to build testing capability and processes that are flexible, adaptable and effective in improving software quality.

Marie currently holds the vice chair position on the Australia and New Zealand Testing Board (ANZTB), a national board of the ISTQB®, and is a co-author of the ISTQB® Foundation Level Agile Tester syllabus.

# **FOREWORD**

I am honoured to write this foreword for a book that I am confident will become a milestone in the testing domain and a reference for the Agile community.

A recent survey by the ISTQB® gathered feedback from over 3,300 respondents world-wide. In this survey, Agile was indicated as the most widely used software development lifecycle (SDLC), well over the traditional waterfall approach and any other development approach. This gives clear evidence that Agile development is not a temporary way of working, but is here to stay.

For this reason, we need to have sound supporting processes for our Agile projects, starting with quality assurance and testing. Quality assurance and testing activities are strongly intertwined with design and development activities. Quality assurance, testing and development activities need to be managed in a well-defined and coordinated way to retain the flexibility and effectiveness that have marked the success of Agile. In addition, providing good coordination between quality assurance and testing on the one hand and development on the other supports the level of quality required for business-critical software. Such coordination is also important to support other challenges, such as usability, mobile channels and security, just to mention a few.

Moreover, we need to understand how to manage Agile ‘in the large’, applying these methodologies to large, complex projects. Such projects are often characterised by large, geographically distributed teams. Such projects require integration testing at several levels.

The need to adapt testing to the Agile paradigm is also witnessed by the remarkable success that the Agile Tester syllabus has obtained worldwide. As of June 2016, just two years after its approval by the ISTQB® General Assembly and just 18 months after its global release, the Agile Tester examination has been taken by over 6,500 professionals, an increase of 60 per cent between the first half of 2015 and the second half of 2016.

Quantitative data clearly show that the ISTQB® Agile Tester syllabus has become a reference in the testing community. As of November 2016, it has been freely downloaded by more than 125,000 professionals!

Therefore, I am sure that many people will greatly appreciate this book, written by some of the authors of the Agile Tester syllabus. This book includes additional topics, extensive examples and useful references that will greatly help both those that want to use it as a study guide to best prepare for the ISTQB® exam and those that simply want to get a better understanding of how to structure testing activities in an Agile realm.

At the ISTQB® we have decided to further invest in Agile testing, dedicating a full stream of our renewed product portfolio to this topic. In addition, we have started the development of new syllabi at the advanced level, covering both technical and organisational aspects of Agile testing in more details.

So, this is just the beginning ... stay tuned!

**Gualtiero Bazzana**

ISTQB® President

# **ACKNOWLEDGEMENTS**

The authors wish to acknowledge the International Software Testing Qualifications Board, and especially the members of the Agile Working Group who created the syllabus upon which we based this book. Many of those authors chose to participate in the writing of this book, but, in a way, all syllabus authors and reviewers contributed.

We also acknowledge the original signatories of the Agile Manifesto, who disrupted but also reinvigorated software engineering. By mixing new ideas with established best practices, Agile methods have made software faster-moving and more dynamic.

Most of all, we acknowledge the day-to-day practitioners of software testing in Agile teams. The initial ideas of Agile methods did not focus on software testing as a special profession – which it certainly is – but these practitioners, through their work in the trenches, have adapted and developed proven best practices of testing in Agile teams. This book is dedicated to them, and we hope it helps to advance the work of testers who provide an essential service to Agile teams.

# ABBREVIATIONS

<b>ATDD</b>	acceptance test-driven development
<b>BACS</b>	bankers automated clearing services
<b>BDD</b>	behaviour-driven development
<b>BVT</b>	build verification test
<b>CASE</b>	computer aided software engineering
<b>CI</b>	continuous integration
<b>CIS</b>	continuous integration system
<b>CPU</b>	central processing unit
<b>CRC</b>	class, responsibilities and collaboration
<b>CRUD</b>	create, read, update, delete
<b>CTFL</b>	Certified Tester Foundation Level
<b>DoD</b>	definition of done
<b>DoR</b>	definition of ready
<b>FDA</b>	Food and Drug Administration (USA)
<b>GUI</b>	graphical user interface
<b>ISTQB®</b>	International Software Testing Qualifications Board
<b>JIT</b>	just in time
<b>MBT</b>	model-based testing
<b>MC/DC</b>	modified condition/decision coverage

<b>MVP</b>	minimum viable product
<b>OCL</b>	object constraint language
<b>PBI</b>	product backlog item
<b>QA</b>	quality assurance
<b>QC</b>	quality characteristic
<b>REST</b>	representational state transfer
<b>RUP</b>	rational unified process
<b>SBE</b>	specification by example
<b>SDET</b>	Software Development Engineer in Test
<b>TDD</b>	test-driven development
<b>UAT</b>	user acceptance testing
<b>UI</b>	user interface
<b>UML</b>	Unified Modeling Language
<b>XP</b>	Extreme Programming

# GLOSSARY

These definitions are taken, with permission, from the ISTQB® glossary (<http://astqb.org/glossary/>).

**acceptance criteria** The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorised entity.

**Agile Manifesto** A statement on the values that underpin Agile software development. The values are: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.

**Agile software development** A group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organising cross-functional teams.

**build verification test (BVT)** A set of automated tests which validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g. Agile projects) and it is run on every new build before the build is released for further testing.

**configuration item** An aggregation of hardware, software or both, that is designated for configuration management and treated as a single entity in the configuration management process.

**configuration management** A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.

**exploratory testing** An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

**incremental development model** A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritised and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a mini V-model with its own design, coding and testing phases.

**iterative development model** A development lifecycle where a project is broken into a usually large number of iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product.

**performance testing** Testing to determine the performance of a software product.

**product risk** A risk directly related to the test object.

**quality risk** A product risk related to a quality attribute.

**regression testing** Testing of a previously tested program following

modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

**software lifecycle** The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note these phases may overlap or be performed iteratively.

**test approach** The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed.

**test automation** The use of software to perform or support test activities, e.g. test management, test design, test execution and results checking.

**test basis** All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.

**test charter** A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing.

**test-driven development (TDD)** A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

**test estimation** The calculated approximation of a result related to various aspects of testing (e.g. effort spent, completion date, costs involved, number

of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy.

**test execution automation** The use of software, e.g. capture/playback tools, to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.

**test oracle** A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialised knowledge, but should not be the code.

**test strategy** A high-level description of the test levels to be performed and the testing within those levels for an organisation or programme (one or more projects).

**unit test framework** A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities.

**user story** A high-level user or business requirement commonly used in Agile software development, typically consisting of one sentence in the everyday or business language capturing what functionality a user needs and the reason behind this, any non-functional criteria, and also includes acceptance criteria.

# PREFACE

This book is for all Agile testers. This includes traditional testers who have made the move to Agile projects and whose roles are almost exclusively involved with testing. However, it also includes developers on Agile projects, who must unit test their own code using techniques such as test-driven development. It includes software development engineers in test (SDETs), who are focused on testing, but play a very technical role within Agile teams as well. It includes product owners and other business stakeholders, who participate in the definition of acceptance criteria and the transformation of those acceptance criteria into acceptance test-driven development and behaviour-driven development tests.

This book is appropriate for people with a variety of backgrounds. If you are an experienced tester new to testing in Agile lifecycles, this book should give you a good introduction to how to be effective in Agile teams. If you are completely new to both testing and Agile lifecycles, you can use this book to learn about both, though we recommend that you also read a separate book on the ISTQB® Foundation syllabus to learn more about testing in general. If you are a developer, SDET or product owner looking to understand the testing perspective on Agile, this book is perfect for you, too.

Based on the ISTQB® Agile Foundation syllabus, this book is also ideal for people who intend to study for and obtain the ISTQB® Agile Foundation

certification. All of the authors of this book were also authors of the syllabus, so we have some special insights that we can share with you about the meaning of the syllabus and what to expect in the exam. In [Chapter 5](#) we have included a complete mock exam so you can check to see if you're ready for the real one. The correct and incorrect choices for the mock exam questions are included and fully explained.

However, it is not enough to pass an exam. You need to be able to apply these concepts in the real world. Within each chapter, we have included sample exam questions for every learning objective, so that you can practice as you go, together with exercises for the key concepts. You will find the solutions for these sample questions and exercises in the Appendix. You should be sure to work through these exercises, because doing so will make you a better Agile tester.

Throughout this book, we have followed the structure and flow of the syllabus, which was deliberately chosen to answer the following questions:

- [Chapter 1](#): What is Agile development?
- [Chapter 2](#): How does testing fit into Agile development?
- [Chapter 3](#): As an Agile tester, what specifically do I need to do?

In other words, we start at the highest level and then proceed to become more and more detailed as we go on.

Each section within each chapter was written by a different author. So, you may detect some difference in tone from one section to another, but we have taken steps to make sure that we are consistent in terms of detail, presentation, full coverage of the syllabus, depth of material and conceptual difficulty. Each author selected his or her section carefully, based on their practical expertise with the topics covered in it. Therefore, the slight tone shifts from one section to another indicate a hand-off within the author team to the person best able to present the topic.

# **1 AGILE SOFTWARE DEVELOPMENT**

by Gerry Coleman and Bertrand Cornanger

In this chapter, we will look first at the concepts and fundamentals of Agile software development. We discuss Agile values, principles and the Agile Manifesto with a particular focus on the advantages of the whole-team approach and the benefits of early and frequent feedback.

We next explore a number of different Agile approaches and examine key practices, including writing testable user stories in conjunction with other Agile team members, and look at how retrospectives can be used as a process improvement approach in Agile projects. In addition, we describe how Agile teams use continuous integration and reflect on the differences between release and iteration planning and how a skilled tester can add value to these activities.

As the demand for new technological products grows apace, many companies have adopted Agile methodologies for their software development activity. The demands on a tester working on projects using Agile methodologies are different from those within a traditional software project. Testers on Agile projects, in addition to excellent testing skills, must be able to communicate

effectively, interact and collaborate constantly with colleagues, liaise closely with business representatives, and work as part of a whole team responding to early and frequent customer feedback.

To work as part of an Agile software team, testers must first understand the principles and values underpinning the Agile development approach. This chapter looks at the essential components of Agile software development, explains the advantages of the whole-team approach used in Agile projects and demonstrates the benefits of receiving early and frequent customer feedback.

At the start of each section, we will introduce the learning objectives for that section from the ISTQB® Agile Tester syllabus. These learning objectives are the basic concepts of the section, explained in the section itself, and are the basis of the questions you can expect to find in the exam for each section, should you choose to take it. You can take a look at the syllabus here: [www.istqb.org/downloads/syllabi/agile-tester-extension-syllabus.xhtml](http://www.istqb.org/downloads/syllabi/agile-tester-extension-syllabus.xhtml)

## **1.1 THE FUNDAMENTALS OF AGILE SOFTWARE DEVELOPMENT**

The learning objectives for this section are:

FA-1.1.1 (K1) Recall the basic concept of Agile software development based on the Agile Manifesto

FA-1.1.2 (K2) Understand the advantages of the whole-team approach

FA-1.1.3 (K2) Understand the benefits of early and frequent feedback

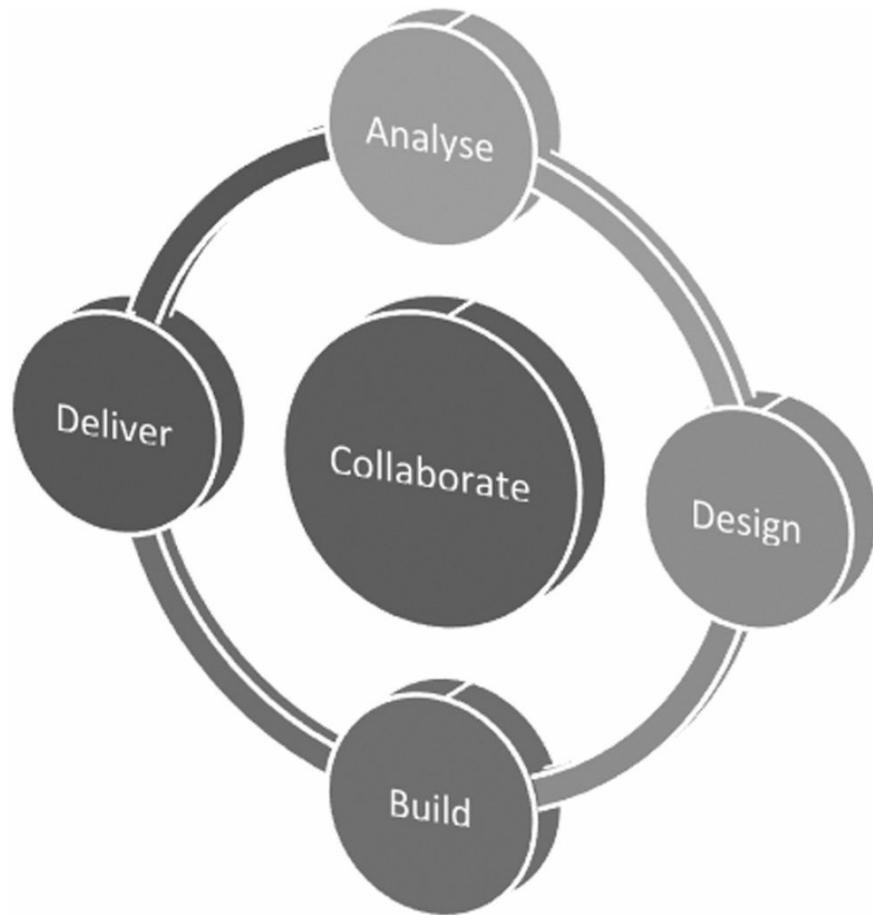
In order to function effectively as a tester on an Agile project, you first need to understand how Agile software development works. Agile development, which covers a range of development models, is based on an ‘iterative’ approach to creating software. In this way, the software product is built in

progressive small chunks with each new piece adding to the features developed in the preceding iteration.

Agile development focuses on early delivery of working software and aims to get the product into the hands of the customer as quickly as possible. Not only does the customer now receive a working product sooner in the development cycle, they can also provide early feedback on features, elements in the product which they like/dislike, and aspects of the solution that they wish to remove or modify. In this way, Agile development can improve customer satisfaction and produce solutions that more closely meet customer needs. A simple Agile development process is shown in [Figure 1.1](#).

---

**Figure 1.1 Overview of the Agile development process**



### **1.1.1 Agile software development and the Agile Manifesto**

As Agile development emerged from a number of different models, a loose conglomerate of the model creators and protagonists convened with a view to unify the various approaches around a common philosophy of shared principles and values. The outcome of these discussions was the Agile Manifesto (Agile Manifesto, 2001). The Agile Manifesto (see [Figure 1.2](#)) contains four statements of values.

Agile proponents emphasise adhering to the values on the left as they believe they have greater value than those on the right.

---

**Figure 1.2 The Agile Manifesto key values**

- **Individuals and interactions over** processes and tools
- **Working software over** comprehensive documentation
- **Customer collaboration over** contract negotiation
- **Responding to change over** following a plan.

#### **Individuals and interactions**

If we take the first Agile value statement, Agile projects encourage communication between team members and between the development team and the customer. In traditional software development projects, the emphasis is often on ensuring that the right processes are followed and that deviations from the process are detrimental and should not be encouraged. In addition, it is often argued that tools are seen as a solution to log-jams that often occur in traditional software development lifecycles.

In Agile the focus is on the team members and their abilities. Confidence in the team to deliver is enhanced by the team's ability to interact. Testers in an Agile project are an integral part of the development team and it is through

working together as a single unified team that product delivery targets are met. Feature-rich, expensive tools are not necessarily the key to success. With the right team, even the most ‘lo-fi’ tools, such as whiteboards and sticky notes, can be significantly more effective than mandating that an underperforming team use a computer aided software engineering (CASE) tool.

## **Working software**

One of the criticisms made of traditional approaches to developing software is the strong emphasis that is placed on detailed documentation covering the outputs of each development phase. Agile development, by contrast, specifies that ‘just enough’ documentation should be produced and that the focus instead is on creating the working product early and often. The most important deliverable of any software project is the product itself. Agile prides itself on regular delivery of the desired features to the customer.

Rather than the Agile team spending time producing detailed requirements specifications, design documents, test plans and so on, the focus is primarily on creating the solution that the customer desires. In teams that create lots of documents, a huge challenge is to keep the documents up to date, particularly where customer requirements are constantly changing or there is uncertainty in what the final product will look like. Agile teams, by contrast, create ‘just enough’ documentation.

By transferring the emphasis from producing documentation to creating a product, communication and interaction between all team members becomes essential. Without reports and templates being the repository of agreement, knowledge is shared verbally and the relationship with the customer deepened and enhanced.

## **Customer collaboration**

Too often in software development a document, such as a requirements specification, becomes the de facto contract between the software development team and the customer. As a contract, everyone is focused on getting this document watertight, removing any ambiguities and achieving sign-off between both parties. This is typically done before the project starts.

In many cases the next involvement the customer has is when the system is delivered and they are asked to conduct acceptance testing to confirm that what they specified in the contract is what has been delivered. When, as often happens, there are great divergences in what the customer believes they are getting and what they actually receive, the requirements specification is revisited and both parties may enter a confrontational phase over what the ‘contract’ actually states.

Agile avoids this process by enshrining customer collaboration as one of its values. Through working closely with the customer, and delivering the software on a regular basis, the Agile team gets constant feedback and any change requests can easily be incorporated before too much additional work is done. Treating the project sponsor as a partner in the project is a key tenet of Agile development.

## **Responding to change**

Who has worked on a software project where customer requirements did not change during the project? It would be incredible if, over even a three to six-month development timeframe, customers did not wish to modify existing requirements, or incorporate new features into the product. Being able to respond to change in this way is essential for happy customers. Too many projects spend large amounts of time at the outset on creating detailed plans. As these projects progress, things happen that affect the project: the customer’s business environment changes; a competitor launches a similar product; new legislation is enacted that will affect the product substantially.

Any number of unforeseeable events can occur, which means the project must be easily able to accommodate changes. Agile, through the use of iterative development, focuses on general long-term release planning and short-term detailed planning. It is impossible to say what may happen during the product lifecycle, so being able to respond to change quickly and effectively can be the difference between project success and failure.

### **The Agile principles**

The four Agile values drove the creation of 12 principles (see [Figure 1.3](#)), which sought to differentiate Agile projects from traditional projects.

---

### **Figure 1.3 Twelve principles of the Agile Manifesto**

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, at intervals of between a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximising the amount of work not done – is essential.
11. The best architectures, requirements and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Let us look at these principles in more detail.

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*  
Agile teams endeavour to deliver software on a frequent basis. Each version of the software includes additional features from the preceding version.
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*  
In Agile projects, change is considered inevitable. Change is seen as positive in that the customer, through seeing features early, has a better understanding of what they want.
3. *Deliver working software frequently, at intervals of between a couple of weeks to a couple of months, with a preference to the shorter timescale.*  
Giving the customer even a rudimentary solution early in the project starts the feedback process. This is then supplemented with a continuous supply of updated feature requests thus deepening and enriching the customer's feedback.
4. *Business people and developers must work together daily throughout the project.*  
Agile projects depend on the continued involvement of the customer. Having a customer representative on-site during development is the ultimate goal.



5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

People are the key to the success of Agile projects, so remove any impediments to productivity and ensure the team is empowered to make decisions.

6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

The primary method of exchanging information within an Agile team is through verbal communication, not documentation. It is impossible to capture every nuance of a feature in a document. Face-to-face communication clarifies requirements and helps to eliminate ambiguity. Where the nature of the project makes face-to-face communication difficult, such as in distributed development, Agile projects will then utilise web/videoconferencing, instant messaging and telephone calls as substitutes.

7. *Working software is the primary measure of progress.*

Rather than use phase completion as a yardstick for how much headway a project has made, Agile development measures progress by how much functionality is working. This introduces the issue of 'definition of done' (see Section 1.2), whereby features are only said to be complete when they have passed customer acceptance tests.

8. *Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.*

Excessive overtime and continual pressure can result in mistakes, omissions and, ultimately, employee burnout and disaffection. Agile teams aim for a sustainable pace that minimises overtime and excessive hours. This approach ensures motivated team members and high-quality work.

9. *Continuous attention to technical excellence and good design enhances agility.*

To achieve a high-quality end product requires a focus on quality all the way through development. Agile teams use good design techniques and issues like redundant code are dealt with as they arise.

10. *Simplicity – the art of maximising the amount of work not done – is essential.*

Developing only the solution required to achieve the project goals is a key factor in Agile projects. Agile teams do not try to incorporate every potential future requirement, or possible request, in the initial design. Instead, they focus on making the design as simple as possible. Thus, if the customer wishes to add a new feature, this design simplicity means the request can easily be accommodated.

11. *The best architectures, requirements and designs emerge from self-organising teams.*

An Agile team is an empowered team. The team self-organises and decides how best to achieve project objectives. All team members share responsibility for the success of the project.

12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.*

The best software teams frequently engage in self-reflection. Regularly examining what is working and what is not working allows Agile teams to remain Agile!

### **1.1.2 Whole-team approach**

Agile software development is conducted by cross-functional teams of people who possess a variety of skillsets. The teams typically house a range of expertise including programming, testing, analysis, database administration, user experience and infrastructure among other specialisms. All of this expertise is needed to deliver the product and is executed in Agile projects by virtue of a whole-team approach.

#### **Advantages of using the whole-team approach**

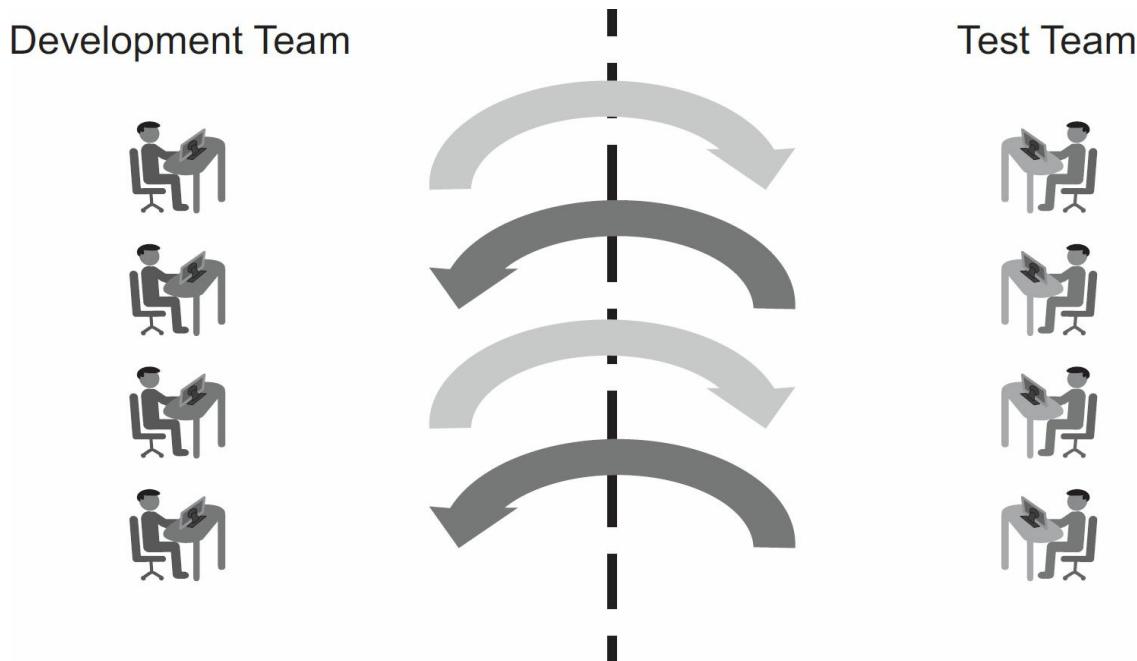
One problem that can occur in traditional lifecycles is a siloing of responsibility, which can mean that essential activities happen too late or that ownership of these activities is unclear. The whole-team approach can help with this. The following subsections explain the advantages of this approach.

##### ***Advantage 1: Making quality everyone's responsibility***

In traditional sequential software development, team roles are very clearly delineated. Quite often boundaries form between the various roles. This results in regular product handover from one group to another, for example from developers to testers. With this product handover can also come responsibility handover. So, in this instance, when code is given to the tester by the developer it now becomes the tester's responsibility rather than the developer's. When the tester finds failures in the delivered software, the product is handed back to the developer to be repaired. Thus, the code is again now the responsibility of the developer (see [Figure 1.4](#)).

---

**Figure 1.4 Transfer of responsibility from developer to tester and vice versa**



All of this ‘back and forth’ activity can easily create division within software development teams. Animosities can arise and testers are often seen as the ‘quality police’ who are the conscience of the cavalier developers!

Agile development approaches software production differently. Within Agile, timely product delivery is the responsibility of the whole development team. The Agile team comprises all of the skills needed by traditional software development teams. However, by creating one team with shared responsibility for quality and delivery, the ‘walls’ that can divide the various roles are broken down and collaboration becomes a necessity. Agile teams are therefore said to be ‘cross-functional’ in that many different roles and specialisations come together in a single team. Combining all of the necessary skills together in a cross-functional team means that the project benefits from the harnessing of this expertise. In Agile the whole (team) can truly be said to be greater than the sum of the parts.

### ***Advantage 2: Enhancing communication and collaboration within the team***

Where a cross-functional team is employed on an Agile project, collaboration

is key to success and is a fundamental component of the whole-team approach. To support close collaboration, Agile teams are generally co-located. Having everyone share the same workspace makes informal contact straightforward. This is enhanced where the customer, or customer representative, is also on-site.

Formal meetings and documentation are minimised in an Agile project, so testers, developers and other team members communicate constantly. One of the few formalities in an Agile project is the stand-up meeting (see [Section 2.2.1](#)). Taking place daily, the stand-up meeting involves the whole team and is used for status reporting of work. By using this simple approach, each team member is kept up to date on what everyone else is doing. However, the stand-up meeting does not preclude the need for individual face-to-face communication by team members. Face-to-face communication helps to remove the impediments highlighted in the stand-up meeting and ensures project momentum is maintained.

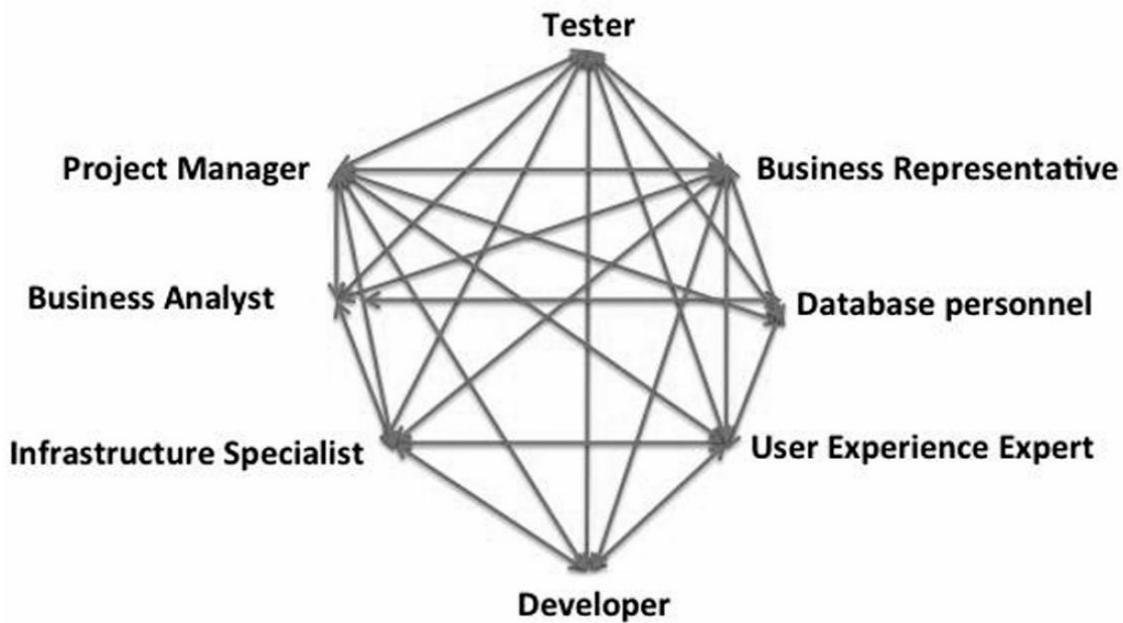
Testers will still need to speak to business representatives, however; and business analysts may need to speak to user experience experts; network specialists may need to speak to developers. The potential communication channels are many and varied (see [Figure 1.5](#)). All of this communication not only helps to share understanding about the product, but also helps to share responsibility for it too. Agile projects benefit greatly from this communication and collaboration.

Agile projects deliberately encourage collaboration through the use of user stories (see [Section 1.2](#)), which are very short descriptions of system features. User stories provide just enough information to allow planning and estimation of work to take place, but not enough for the feature to be developed and tested. For this to happen, it is necessary to speak to the person from the business side who is proposing the feature. Only they, as the customer or customer representative, know exactly how the feature should

work, so collaborating closely with them greatly aids in understanding the requirement.

---

**Figure 1.5 Potential communication channels in an Agile team**



***Advantage 3: Enabling the various skillsets within the team to be leveraged to the benefit of the project***

Testers on an Agile team work with the business representatives to develop suitable user acceptance tests. Only the business representative will know how exactly they wish the system to function. Working together with the business sponsor, the tester can assist in creating acceptance tests that help the team to determine when a system feature is complete or ‘done’. In Agile projects, many of these acceptance tests are automated, so there is great onus on the tester to ensure feature requests from business representatives are as testable as possible and can be easily encoded into a tool.

As part of their role within the whole-team approach, testers will also work closely with developers. Testers can help developers to create automated unit tests, but have an even greater role to play where continuous integration (CI)

is used (see [Section 1.2](#)). Daily, or more frequently, builds of software demand continuous regression testing. For CI to be effective, the regression tests must be automated. Working together, developers and testers can utilise their skills to create an automated regression test suite that is used each time a new system build is created. In addition, the test suite must be capable of being easily modified and added to as new functionality is integrated, and thus new corresponding tests are required.

Through collaboration with business representatives and developers, testers can help to share knowledge about testing. Cross-team working increases knowledge sharing and all team members get a better understanding of each other's work.

One of the drawbacks with traditional software development projects is that developers and testers frequently have no interaction whatsoever with the customer. Much of the product information they receive is through project documents or from conversations with other development team members such as business analysts. Encouraging testers and developers to collaborate directly with customers or customer representatives means system features are much better understood than would otherwise be the case.

Crispin and Gregory (2009: 24) propose using the ‘Power of Three’ for such team interactions. Any feature meeting or discussion should involve a business representative, a developer and a tester. The presence of all three roles in discussions provides a shared clarity and understanding of system features within the team.

### **1.1.3 Early and frequent feedback**

In sequential development projects, the development cycle can be long, and customer involvement in the process may be limited, with most feedback coming late in the process. Agile development benefits from early and frequent feedback from the customer.

Benefit 1: Avoiding requirements misunderstandings, which may not have been detected until later in the development cycle when they are more expensive to fix.

In software projects that use traditional methods, customers are heavily involved in the early phases of the process during requirements capture and definition. Once the customer has signed-off on the requirements document, the next major input they may have is at the later stages of product testing when it is too late to remedy requirements misunderstandings or misinterpretation. It is also very expensive to make changes at this stage, as the amount of re-work may be substantial. Continual customer involvement resulting in early and frequent feedback helps to eliminate requirements ambiguity.

Benefit 2: Clarifying customer feature requests, making them available for customer use early. This way, the product better reflects what the customer wants.

Agile projects use short iterations during the development cycle. Within these short cycles, feature clarification, design, code and test takes place. At the end of the cycle, the team have produced a ‘minimum viable product’ (MVP). The MVP contains a very limited subset of proposed system features, but is sufficient to enable meaningful feedback to be gathered from the product’s initial users. Each subsequent development iteration then adds to the product until, eventually, the entire system is complete. Business representatives on the project assist in feature prioritisation, resulting in the most important features, those with the highest business value, being developed first. This also reduces system risk, as we always know we are working on the system aspects that have greatest customer worth.

Benefit 3: Discovering (via continuous integration), isolating and resolving quality problems early.

Projects that use sequential software development approaches perform

integration testing towards the end of the development lifecycle. Any integration problems, or system interface issues, only become apparent late in the process and can be a major headache to fix. Using CI, Agile projects benefit from the fact that any integration issues are instantly highlighted, the module(s) concerned can be isolated pending repair and the system can be rolled back to the previous working version. When the fix is made, the module can then be reintegrated into the system and re-tested. Quality problems are resolved early before there is additional downstream system damage.

Benefit 4: Providing information to the Agile team regarding its productivity and ability to deliver.

Early and frequent feedback from the customer provides knowledge to the Agile team on its productivity. Each feature is estimated and prioritised prior to development and then built during a development iteration. When the feature passes the acceptance tests it can be classed as ‘done’. This process allows the Agile team to measure how much product it can deliver per iteration. Measures such as these demonstrate the team’s productivity, or ‘velocity’, and greatly assist with subsequent estimation and release planning.

In addition, the Agile team can see what is hindering their productivity and progress, allowing them take remedial action.

Benefit 5: Promoting consistent project momentum.

Project team morale is improved by the visibility of progress. Getting early and frequent feedback from the customer makes all parties more engaged. Satisfying the customer is hugely rewarding for the Agile team.

## **Summary**

Agile development provides an alternative to traditional sequential software

development. Agile promotes the benefits of working software, customer collaboration and the ability to respond to change and values talented individuals and encourages them to interact continually to the benefit of the project. The Agile philosophy is supported through 12 key principles.

By making the customer requirements central, Agile seeks to ensure complete customer satisfaction with the finished system. Agile uses a whole-team approach, whereby everyone has responsibility for quality, not just testers. Quality is improved, and client satisfaction is achieved, by getting early and continuous feedback from the customer.

## Test your knowledge

The answers to these questions can be found in the Appendix.

### Agile Tester Sample Questions – [Chapter 1 – Section 1.1](#)

#### Question 1 (FA-1.1.1) K1

A developer on your Agile team, who has previously spent many years working on software projects using traditional approaches, believes that the system design should try to anticipate all future customer needs. Which of the following Agile Manifesto principles BEST indicates that this approach is not suitable for Agile projects?

#### Answer set:

- A. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
- B. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely
- C. Simplicity – the art of maximising the amount of work not done – is

- essential
- D. The best architectures, requirements and designs emerge from self-organising teams

**Question 2 (FA-1.1.2) K2**

Consider the following guidelines that could be implemented on a software project:

- i. Where possible, restrict the team size to a maximum of 10, with use of additional technical experts only as required.
- ii. To save time, only have team members attend daily stand-up meetings when they have something new to report.
- iii. Use the ‘power of three’ concept for all user story discussions.
- iv. Locate the development team and test team in separate offices so that testing remains independent.

Which of the above guidelines should be used in an Agile project using the whole-team approach?

**Answer set:**

- A. (i) and (ii)
- B. (iii) and (iv)
- C. (i) and (iii)
- D. (ii) and (iv)

**Question 3 (FA-1.1.3) K2**

Three months into an Agile project, a customer is unhappy that they have not seen any evidence of working software nor been approached to provide any feedback. Which of the following might have prevented this situation?

### **Answer set:**

- A. The use of short development iterations to build the software
- B. The use of the ‘V’ model from the commencement of the project
- C. Delaying the start of the project until all system requirements were clearly understood
- D. The use of web conferencing tools to simplify communication between the customer and the development team

## **1.2 ASPECTS OF AGILE APPROACHES**

The learning objectives for this section are:

FA-1.2.1 (K1) Recall Agile software development approaches

FA-1.2.2 (K3) Write testable user stories in collaboration with developers and business representatives

FA-1.2.3 (K2) Understand how retrospectives can be used as a mechanism for process improvement in Agile projects

FA-1.2.4 (K2) Understand the use and purpose of continuous integration

FA-1.2.5 (K1) Know the differences between iteration and release planning, and how a tester adds value in each of these activities

The market is full of a range of models aimed at supporting those who wish to move to Agile development. These include methodologies from helping to manage Agile projects, to supporting practices that can be used by Agile teams. Some of the more popular Agile methodologies are looked at here. They include Extreme Programming, Scrum and Kanban. While there are subtle differences between these methodologies, they have many features in common.

User stories are used as a way to capture system requirements, and are written collaboratively by developers, testers and business representatives. Continual

process improvement is a hallmark of Agile development, and the featured models use retrospectives to examine how productivity and effectiveness can be optimised. Each of the models use CI to ensure a high quality of software from the outset, and release and iteration planning is used to guide the project forward. The process of release and iteration planning is a detailed and complex one and provides significant opportunity for a tester to add input, expertise and value to this part of project management.

### **1.2.1 Agile software development approaches**

Agile is not a unique methodology and there are multiple ways of implementing it. The most widely known Agile development methods are Extreme Programming (XP), Kanban and Scrum and usage of these often depends on the country and project type.

As they are not addressing the same topics, some projects implement several techniques. For example, Scrum may be used with XP for a new project, whereas XP and Kanban may be deployed for software maintenance. Some Agile methods have similar practices, for example the concept of visual management using boards, or the idea of a backlog, which is explained in the Kanban subsection.

#### **Extreme Programming**

Extreme Programming (XP) (Beck and Andres, 2005), is an Agile approach to software development described by certain values, principles and development practices:

- As reviewing code is a good practice, two people will work together on the review.
- As testing eliminates defects, code will be tested intensively with unit tests before releasing a feature.

As design is important to ensure that changes in code will not impact

- other parts, dependencies are minimised.
- As building and integrating is important, it will occur several times a day.
- As changes in user needs must be taken into account, and occur often, there will be several short development cycles.

The **common activities** of a team implementing XP are:

- Listening: The developer must listen carefully to the customer needs in order to propose a technical solution matching the need.
- Designing: Design is done in order to reduce the risk of regression when new features are added. Throughout the project, the team searches to limit dependencies. The code is continually refactored in order to make it clear and simple. To make it more understandable to all stakeholders, metaphors and analogies are used to describe how the system works. Class, responsibilities and collaboration (CRC) cards are used to describe objects to be implemented. CRC cards show a class (representing a collection of similar objects), responsibilities (attributes of the class, or things the class can do) and collaborators (other classes the class uses to execute those responsibilities). [Figure 1.6](#) shows an example of a CRC card for a university student.
- Coding: Core in the process, coding is the main activity, because without code there is no software. Coding is pair programmed and must be compliant with the development company's coding standards.
- Testing: Testing is approached from three different perspectives with the emphasis on automation:
  - Unit testing: Using test-driven development (see [Section 3.1.1](#)) to

ensure that the feature is correctly developed.

- System wide integration: To check the correct functioning of the interfaces in the system.
- Acceptance testing: To ensure that the customer needs have been correctly implemented.

---

### **Figure 1.6 Example of a CRC card for a university student**

---

**Class:**

*Student*

Responsibilities:	Collaborators:
<i>ID Number</i>	<i>Module</i>
<i>Name</i>	
<i>Address</i>	
<i>Email</i>	
<i>Selects module options</i>	
<i>Enrols on module</i>	
<i>Takes exam</i>	

---

Even if project management is not the main purpose of XP, planning and managing can be added to the XP activities.

- Planning: The project is divided into iterations. User stories, which describe what the system must do, are used to estimate the implementation effort. Each iteration produces small units of functionality to meet the business requirements.
- Managing: One open space is used for the team members, stand-up meetings occur daily and project velocity is measured to help achieve a sustainable pace.

**Five values** guide development with XP:

- Communication: Daily face-to-face meetings are organised and all project members are in the same team and work together on all tasks throughout the project. Working together, the whole team will produce better solutions than having divided responsibilities.
- Simplicity: Only what is needed and enough is done for all team activities. This allows for more rapid progress.
- Feedback:
  - From the customer: Acceptance tests are designed by the customer and testers and run to show the current state of the system.
  - From the system: Tests are used to check regression issues.
  - From the team: Estimation of the time needed to implement the requirement.
- Courage: Team members need courage that can be translated into working hard to reach the objective and deliver the predicted features on time. A lot of impediments can occur, such as discovering a piece of code is out of scope because the architecture is overly complex. Sometimes, knowing exactly how to code a feature can be difficult and several prototypes may be required to resolve this.
- Respect: Good team spirit leads to a good work product. All team members must respect each other as everyone works with one objective: delivering a working software product.

XP describes **a set of principles** as additional guidelines:

- Humanity: A software component is developed by humans, for humans. User needs have to be well understood and team members must work in a good atmosphere.
- Economics: As in Lean development, all actions must generate added value.

- Mutual benefit: Everything I do must be useful to others.
- Self-similarity: Try to reproduce effective actions, such as using design patterns for coding, or use similar scripts to create regression tests.
- Improvement: Always seek to improve your practices.
- Diversity: Do not be narrow-minded and always be open to suggestions.
- Reflection: Analyse the process and how well it is working.
- Flow: Work in a continuous flow instead of staggered phases.
- Opportunity: Instead of being viewed as problems, impediments must be seen as opportunities to become better.
- Redundancy: Like the principle Diversity, using different approaches to find a solution is useful.
- Failure: Failures are normal, and success may require multiple versions of prototypes. Different attempts may be needed before you have a working version.
- Quality: Quality must remain a primary objective and should not be compromised.
- Baby steps: Incremental small changes are better than attempting major changes in a short space of time.
- Accepted responsibility: No assignment of responsibility should occur; it must be accepted by team members. A team member accepts a task or a set of tasks and must perform all of them.

XP describes 13 **primary** team working practices:

- Sit together: All team members work in the same open space to improve communication.
- Whole team: All the skills and competences required for success must

be incorporated into the team.

- Informative workspace: Team progression is visible on walls or boards.
- Energised work: Work only as many hours as necessary to retain maximum efficiency.
- Pair programming: Two people sitting at one machine should be used for all programming.
- Stories: Simple and clear stories make estimating simpler. They are discussed with the customer representatives during estimation meetings.
- Weekly cycle: A week is a natural time unit for a team. A week may begin with an open meeting to get customer stories and report progression. Automated tests are created, then coding is undertaken and the week finishes with the software being deployed.
- Quarterly cycle: During a quarter, a theme of stories is implemented. Planning around themes ensures the team sees the bigger picture and does not spend excessive time on detail.
- Slack: Some slack (small non-important stories) is added as a reserve of time if necessary.
- Ten-minute build: Automatically building the system, and running tests, in 10 minutes is the optimal goal and provides almost instant feedback.
- Continuous integration: Changes are integrated and tested every two hours.
- Test-first programming: Automated unit tests are written before coding.
- Incremental design: Design only that which is necessary and allow for future changes.

Most of the Agile software development approaches in use today are influenced by XP and its values and principles. For example, Agile teams following Scrum often incorporate XP practices.

## **Scrum**

Scrum is an Agile process dedicated to team organisation and project management. Its goal is to improve team efficiency. As an Agile project framework, Scrum provides roles, iterations, specific meetings, artefacts and rules. Scrum does not mandate what practices to use, so the team must decide their own approach to doing things. Depending on the context of the project, Extreme Programming is often the methodology used to support the Scrum framework.

As a team with inexperienced people can often make mistakes early in a project, so a coach should be present from the beginning. An experienced Scrum master will be able to motivate a team in all its activities. To implement Scrum, you will need at a minimum:

- A wall where you can place ‘sticky notes’, representing the project stories, tasks and impediments, thus providing a view of the progression of the work.
- Blank sticky notes and pens.
- A set of cards to allow the team to play ‘planning poker’ (see [Section 3.2](#)) to estimate the implementation effort.

Scrum defines three main project roles:

- Scrum master: A member of the Agile team, they ensure that Scrum practices and rules are implemented and followed. The Scrum master resolves any violations, resource issues or other impediments that could prevent the team from following the practices and rules. For example, they record the actions from the different meetings, such as

the daily Scrum, and ensure that each person has a chance to speak. This is a coaching role more than a team leader role, as there is no team leader in Scrum. This requires a change of management style. The Scrum master must support the Scrum team in self-organising and facilitates the team in selecting the backlog items for a sprint.

- Product owner: The product owner is the customer representative on the project. They work in close conjunction with the team. They have the vision of the product to be developed, and of the customer's conditions of satisfaction for each of the user stories. They are mainly responsible for deciding feature criticality and quality characteristics and validate the product once built. They generate, maintain and prioritise the product backlog.
- Scrum team: Self-organised, the team, ideally comprising three to nine members, is in charge of identifying the architecture and developing the code to convert the customer needs into working software. Each team member may have specific skills, such as development, architecture and design; however, as they work together on the same goal, the team is cross-functional. Testing is also a responsibility of the whole team. To succeed, different profiles can belong to the team, architects, developers, testers and so forth and people may join the team, when needed and leave when their tasks are done. For example, a performance testing specialist could join the team and prepare and execute performance tests when there are enough increments to allow it, but remain outside the team otherwise.

There are certain keywords that have specific meanings in Scrum:

- Sprint: Releases are split into iterations (called sprints), which are periods of fixed length (usually two to four weeks).
- Velocity: In simple terms, velocity is a measure of how much work a team can do in a sprint. In Scrum this may be measured as the number

of story points completed (see [Section 3.2.2](#)).

- Product increment: Each sprint results in a potentially releasable/shippable product called an increment. A difference between this and XP is the size of the increment, which can be bigger in Scrum. An increment must contain added value in terms of benefit for the customer.
- Product backlog: From a vision that describes the product to be released, functional and non-functional requirements, or features, are identified and refined until a level of granularity is achieved that allows the team to estimate the effort required for implementation. These features, which comprise Scrum's product backlog, are called product backlog items (PBIs), or user stories, and are implemented in releases and sprints. The PBIs are prioritised, and the backlog is managed by the product owner. The product backlog changes from sprint to sprint during the release planning and grooming sessions, but no changes are allowed during a sprint.

If the team considers that an item is not 'ready' to be implemented, because there are technical issues, or the item is considered too large or needs examples/tests, a lower-ranked user story will ascend in the product backlog.

- Sprint backlog: The product backlog is the source of the sprint content. As the product backlog is prioritised, the highest priority items are placed on top of the list. During the sprint meeting, the team creates the sprint backlog by selecting a set of high priority items from the product backlog. Each sprint has a goal, and the selected user stories must match with this goal in order to create a value-added increment. The stories in the sprint backlog may be divided into tasks to be executed by the team. The Scrum process, illustrating how sprints are incorporated, is shown in [Figure 1.7](#).
- Definition of 'done': In Scrum, a product increment is only

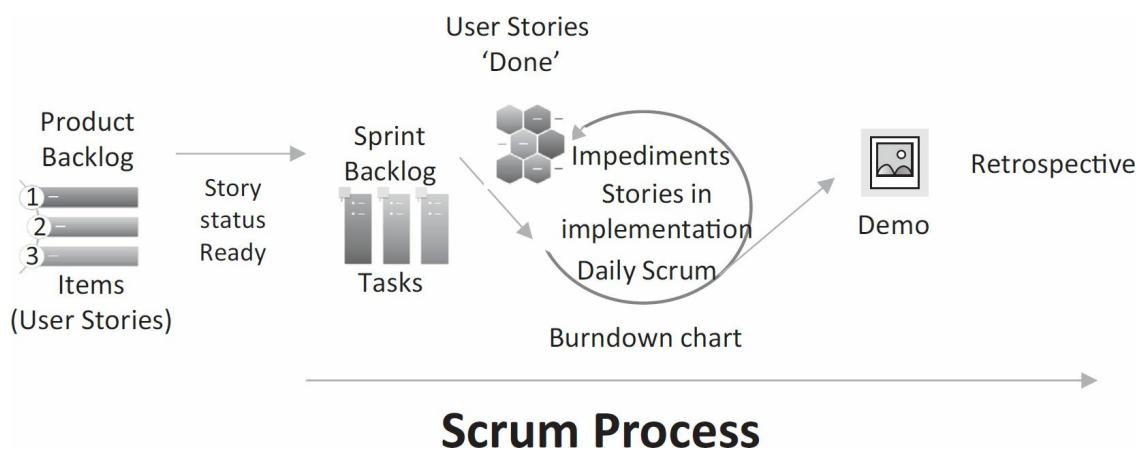
considered ‘done’ if it is potentially shippable and of sufficient quality. Even if the goal of the team during a sprint is always the same, to be able to release a potentially shippable product, practices may differ from one team to another. For example, one team may use test-driven development (TDD) whereas another is unable to use it because that team is developing a system where test automation cannot be performed. So, defining when a story is ‘done’ can change, depending, for example, on the criticality of the project, the team knowledge (business and techniques) or the technology. The definition of ‘done’ may involve several types of test, from unit testing only, to a combination of unit, integration, system and acceptance tests. For a sprint, the success of performance tests may be the main criteria for the stories to be considered ‘done’. This is discussed and defined by the team during the sprint planning meeting. Other statuses may exist for an item in the backlog. For example, the definition of ‘ready’, which defines when a user story is ready to be implemented, or ‘shippable’, which marks a developed item as ready for release.

- Time boxing: The concept of time boxing obliges the team to estimate correctly the time needed for the implementation tasks. The time needed to develop, test and build user stories, the time needed for meetings and the time for studying spikes is estimated to fit in a sprint. Some slack is given by adding non-critical user stories to the sprint. If time is short, these non-critical stories will be moved back to the product backlog for inclusion in a later sprint to allow the team to focus on value-added user stories in the current sprint.
- Transparency: Transparency between team members is important and also ensures the stakeholders have confidence in the team. Transparency allows the team to explain the time they need, how they work, the difficulties they encounter and the quality of the product

they are building. The burndown chart, the list of impediments, the backlog and the board where information is written are all examples of transparency. Transparency is also important during a retrospective to improve the process.

- Daily Scrum: The daily Scrum, or daily stand-up meeting, occurs every day when all the team members are present. They discuss their work and list their project impediments and the upcoming tasks to be done. It is a mechanism for progress reporting and usually lasts about 15–20 minutes.
- Burndown/burnup chart: Both these charts are closely associated with Scrum. Sprint burndown charts and release burnup charts show how the team is progressing against its forecasts. The differences between the forecasts and the actual are analysed during the project, especially in a retrospective, and this analysis allows the team to find solutions to increase its velocity.

**Figure 1.7 Example of a Scrum process**



## Kanban

Like Scrum, Kanban is an Agile management process. The name Kanban means ‘label’ or ‘ticket’. This comes from the industries that need to have a

just-in-time (JIT) flow of components or items available with which to build a product. There are baskets with all these items, which are conveyed to an operator who performs a task on the item, adding value, and then puts the new built item in another basket, which goes to another station where another operator will perform a task on that item, and so on, much like an assembly line. Iterations or sprints are optional, but, typically, a continuous flow of work is used. The name of the item is written on each basket.

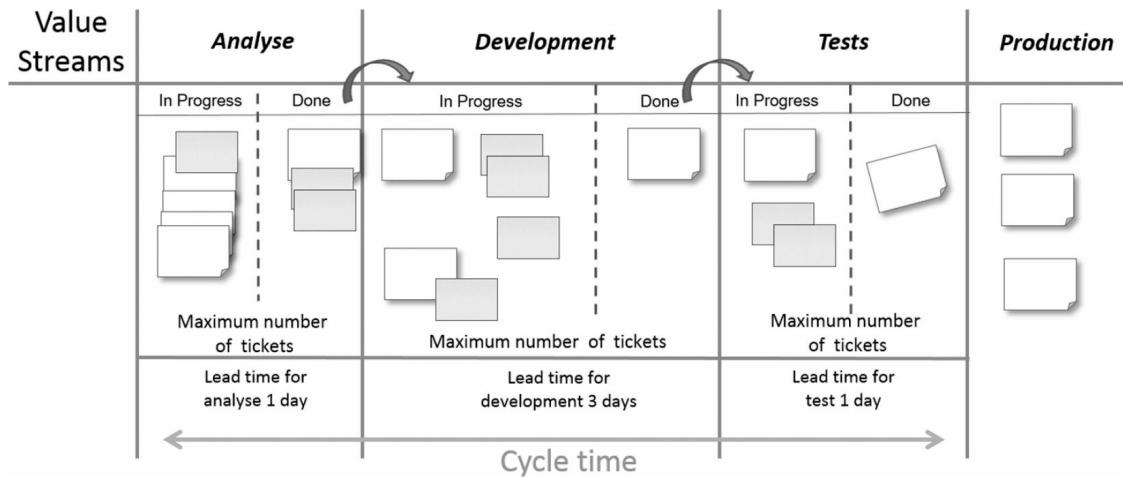
By analogy, an operator is a member of the team who will have an action on an item in a list (basket). Each time a task is performed, the component gains a new status.

Kanban utilises three instruments:

- Kanban board: On a board, several columns list items in different states. Each column represents a station, which is a set of activities. These stations are represented as ‘analyse’, ‘development’ and ‘tests’, as can be seen in [Figure 1.8](#). Here, the items, steps or tasks, symbolised by sticky notes, move from left to right when all activities of a station are done and there is a free slot in the next column.
- Work-in-progress limit: The entire board, and indeed each station, has a limit on the maximum number of tasks it can handle.
- Lead time: Kanban is used to optimise the continuous flow of tasks and the cycle time by minimising the (average) lead time for the complete value stream. So, when a task is finished, the ticket is transferred as soon as there is a free slot in the next station.

---

**Figure 1.8 Example of a Kanban board**



## 1.2.2 Collaborative user story creation

Since the advent of software development, the requirements capture and management has proved especially problematic given software's essentially intangible nature. Though traditional development models have improved their requirement management processes, the Agile refining process can improve this even further. [Table 1.1](#) shows different types of backlog items.

---

**Table 1.1 Types of user stories**

---

<b>Theme</b>	A collection of user stories that have something in common
<b>Epic</b>	A very large user story, which may subsequently be broken down into smaller stories as they get closer to being developed
<b>User story</b>	A feature that the user desires
<b>Spike</b>	A special type of user story used for research or investigation activity

---

In an Agile environment, user stories are the smallest unit of requirements. However, the collaboration between people begins at a high level, before the team refines system features into user stories. At the beginning, the vision of the customer is often translated into themes and epics associated with

customer satisfaction criteria. The product owner can interact with other project stakeholders before involving the team, so that the team begin to work on an already well-defined product backlog (see [Figure 1.9](#)). From this moment, the refining process is used to split features, to make the customer needs more understandable and to better understand technically how to implement them and test them, for example, with spikes. At the user story level, acceptance criteria will be defined to check that the implemented user story matches with the user's conditions of satisfaction.

Each user story must be developed where the team member has a mindset to consider user satisfaction. This explains the recommended format of a user story: ‘As a “type of user”, I want “some goals”, for “some reason”.’ See [Table 1.2](#) for an example of a user story.

**Figure 1.9 Example of product backlog**

Product Backlog										
Id	Id Parent	Associated Theme	User Story	Status	Details	Acceptance Criteria	Régression Tests	Criticality	Size	Sprint
									XL	
1									L	
2		User account							S	
3		Back Office								
4		Shopping	As a customer I want to see a list of all types of washing machines	To be refined	Public, need the database to be populated					
5		Payment	As a customer I want to pay for the items in my basket using Visa to validate my order	In progress	Need to have an account	Test Payment.doc	RT.DOC	1	XL	3

**Table 1.2 User story example**

### User Story:

*‘As an analyst, I want to view data in a statistics tool after extraction so that I can analyse it.’*

The developer will write and test the ‘extract’ function per the functional user story. A test uses the acceptance criteria to check the interoperability and better explain the business need.

### Test:

*Step 1: Ask the production team to launch the batch job extracting data from the field overnight. In the morning check that the batch has finished and returned no errors.*

*Step 2: Ask the analyst to check that the sales completed the previous day are in the statistics.*

Each story will specify acceptance criteria for these functional and non-functional characteristics. These criteria provide the developer and tester with an extended vision of the feature that the product owner and business or operation stakeholders will validate.

---

As users' needs are both functional and non-functional, user stories have to take all quality characteristics (QCs) into account. For example, those listed in ISO 25000 or ISO 9126, including interoperability and usability, can be used to ensure something important, or implicit, for the customer is not forgotten.

During a product risk analysis, a tester may help to make the user needs explicit by checking each QC with open questions during release, iteration, grooming or requirements planning meetings. This approach is particularly useful when implementing acceptance test-driven development (ATDD) (see [Chapter 3](#)).

Risks to be covered by a QC test should be a part of the definition of 'done' for each feature.

## Characteristics of a user story

The INVEST acronym provides a checklist that assists the tester in checking the quality of a user story. INVEST stands for:

- **Independent:** A user story is easiest to develop if it is not dependent on another user story. In this way, a user story can be developed and tested with mocks representing other, yet to be developed, stories.
- **Negotiable:** Initially, the negotiation is about the functionality that the user story has to provide (see [Figure 1.10](#)). How to implement the user story is not discussed at this point.

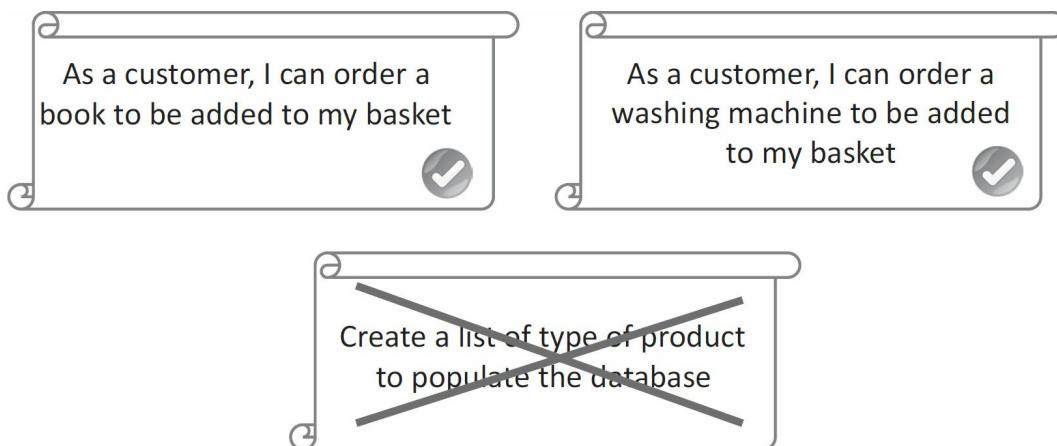
- Valuable:** Implementing a technical feature is not valuable in itself.
- The value comes from the functionality that the user can utilise. In [Figure 1.11](#), we understand that the purpose of the story is to ‘add the order to the basket’.
  - **Estimable:** The user story must be capable of allowing the team to estimate the effort for its implementation. If the user story is too vague, tasks like development or testing cannot be accurately estimated.
  - **Small:** A user story must be sufficiently small so it can be developed in an iteration. Team size and the size/complexity of user stories will help to determine how many stories can be delivered in an iteration.
  - **Testable:** The user story must be testable. If there is insufficient information available about the story to enable tests to be written, then the story should be deferred until such time as that information becomes available.

Another well-known acronym called 3C also helps to characterise a user story:

**Figure 1.10 Stories focus on functionality rather than implementation**



**Figure 1.11 Stories focus on user value**



- **Card:** The card is the note describing the story and contains only basic information. Given the limited detail on the card, the next 'C', a conversation, is required.
- **Conversation:** The conversation is used to achieve a common understanding of the requirement between all stakeholders. The conversation is primarily verbal, but may also be supported by documentation where available/appropriate. For example, the payment feature of a product on a website could generate the following conversation and questions:  
How does the customer pay? By card? What kind of card? Do we have different kinds of customers? How do we test that? Do we have a set of fake payment cards for testing?
- **Confirmation:** All the information collected during the discussion helps to define acceptance tests. Acceptance tests will confirm that the feature is done (see [Figure 1.12](#)).

Using behaviour-driven development (BDD; see [Chapter 3](#)), these tests can also be expressed in the form 'Given-When-Then':

---

**Figure 1.12 Example of tests linked to user stories**

User Story	As a customer, I want to pay for the items in my basket using Visa in order to validate my order
Goal	Check that payment with Visa is a possibility
Notes	
Status	

**Acceptance tests:**

Test Id	Type of test BDD, ATDD, Manual	Description	
		Pre-requisite	Steps
5.1	Manual User: End User	Pre-requisite Steps	The customer must be logged in
			1- Select a product and add it to the basket 2- Click on Payment 3- The price is shown on the screen 4- The customer can select Visa in the list of available payment card 5- It selects Visa and enter its information
5.2	Manual User: End User	Pre-requisite Steps	A message says that Payment is accepted
			No user account for this customer
		Result	6- Select a product and add it to the basket 7- Click on Payment
			An error message asks the customer to be logged in

‘Given I am a customer, when I click on Payment, then I can select Visa in the list.’

‘Given I have selected Visa, when I enter my card details, then my payment is processed.’

Like all documentation, test documentation should be concise, sufficient and necessary. Sometimes, tests are used to help document the project when, in some organisations, detailed specifications are lacking. Tests must also respect the standards governing different industry sectors, such as avionics.

Automated test scripts are themselves based on very detailed test cases. Regression tests also need to be sufficiently detailed to be automatically replayed or executed manually by testers. The information required to create the tests is collected during the conversation.

### 1.2.3 Retrospectives

In Agile development, a retrospective is a meeting, generally held at the end of an iteration, to discuss what was successful in the iteration and what could

be improved. The meeting also focuses on how to incorporate the identified improvements and retain the successes in future iterations. [Table 1.3](#) shows the topics covered by retrospectives.

**Table 1.3 Topics in retrospectives**

Topic	Example
<b>Process</b>	<ul style="list-style-type: none"><li>• Changing the start time of the daily stand-up meeting from 9:00 a.m. to 9:30 a.m. to ensure all team members are present.</li><li>• Deciding to discuss project impediments during the day they arise, not after.</li><li>• Deciding to organise more meetings to discuss the product backlog to avoid ‘surprises’ in how the user story is implemented.</li><li>• Deciding to better define the tasks to be done during sprint. Planning to avoid any potential waste of time.</li><li>Deciding to add the estimation of the test effort, by a dedicated tester, to the size of the user story.</li><li>• Deciding to analyse the root cause of defects and defining the best test types to address these risks.</li></ul>
<b>People</b>	<ul style="list-style-type: none"><li>• Deciding to assign a team member to functional tests.</li><li>• Inviting a performance tester onto the team to define specific test cases.</li><li>• Using a test centre to automate test cases.</li><li>• Using a test centre to organise end-to-end testing without mocks.</li></ul>
<b>Organisations</b>	<ul style="list-style-type: none"><li>• Setting up a Scrum of Scrums and splitting the product backlog across several teams.</li></ul>
<b>Relationships</b>	<ul style="list-style-type: none"><li>• Organising more meetings with user representatives to better understand their needs.</li></ul>

## Tools

- Deciding to run automated test cases as part of the CI process.
  - Storing test cases in a test management tool.
  - Using a tool to create test data.
- 

### Example of the process used for a retrospective

For a three-week iteration, a retrospective meeting of two hours is organised by the Scrum master.

The process is made up of several steps:

- Begin the meeting and set up tools and material
  - It can be useful to create a round table and ask people to score how they experienced the iteration. Each person is encouraged to speak freely.
  - Compare the action list from the previous retrospective with what has been done during this iteration.
- Collect data
  - The list of impediments collected during the iteration is used as the entry point.
  - Ask each team member what has worked well and what was problematic. People can work on positive or negative points using sticky notes.
- Discuss improvements
  - The moderator of the retrospective, for example, the Scrum master in Scrum, must think like a coach. They help people to imagine several solutions, but should not decide how to solve issues.
- Decide actions for implementation of the improvements
  - The team must select one solution from the potential solutions to an issue, and a person responsible for implementing it must be assigned.
  - Not too many improvements should be identified. Step by step, retrospective after retrospective, is an easier way to achieve change.
- Store outputs of the retrospective in an action list and close the meeting/workshop.

To be effective, people must see the decisions made at the retrospective implemented by the team, and logging the results of this meeting is therefore essential. By engaging fully with the retrospective process, the team should be able to identify new ideas instead of harbouring discontent about the

existing situation. Transparency is important, and people must have trust in each other.

Indicators such as the burndown chart, as a mechanism to discuss velocity, are useful. Other indicators, such as test metrics, are also beneficial. An indicator to be followed in each iteration is the number of defects that have to be corrected during the iteration. This indicator will help in making decisions on the test effort required to maintain the team's velocity.

The timing and organisation of the retrospective depends on the particular Agile method followed. It must occur at least at the end of each iteration or when needed. For example, in Scrum, a product demo is held just before the retrospective and the same people attend. With larger Agile teams, there are retrospectives both within the team and across teams.

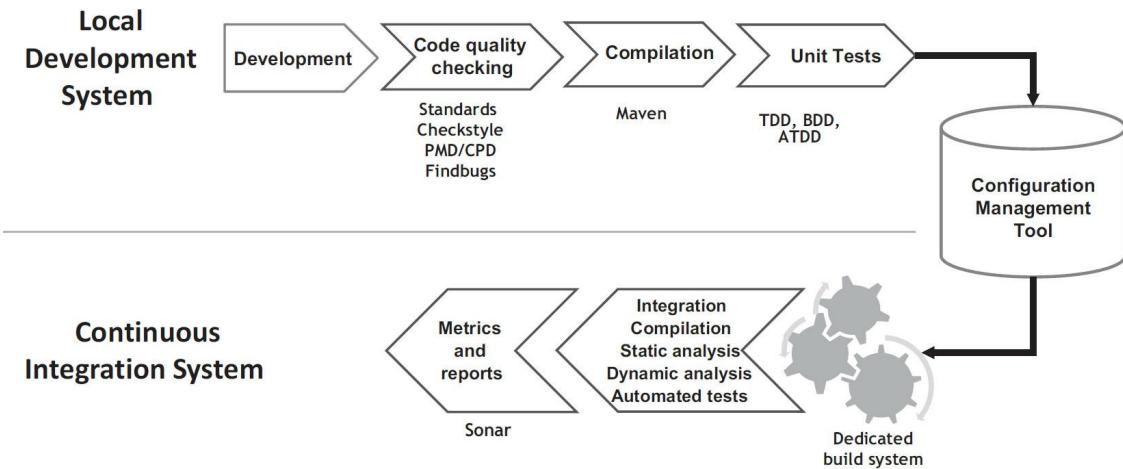
Business representatives and the development team attend each retrospective as participants while the facilitator organises and runs the meeting. In some cases, the teams may invite other participants, such as architects, people from methods or representatives of other Agile teams with user interface expertise, to the meeting.

#### **1.2.4 Continuous integration**

A CI system (see [Figure 1.13](#)), allows the automated compilation, integration, testing and deployment of the entire code base in a project. A variety of tools are used in a CI system, and the code and automated tests are typically placed in a configuration management repository.

---

**Figure 1.13 Example of a continuous integration system**



A process using a CI system may work as follows:

- The developer codes a function, ideally using the test-driven approach. In this way, they develop an automated unit test for the function.
- Static analysers are used to determine compliance with the in-house coding style and complexity metrics may be derived to determine readability. These actions may be executed by the CI system itself.
- Finally, if the developer is satisfied, they place their code and associated automated tests in the shared repository.

During the CI process, all code is extracted, compiled, built and tests executed. Some CI systems create test environments before deploying. Sometimes, to reproduce initial test conditions, databases may have to be reinitialised. Some may also be able to run black-box automated acceptance tests on specific computers remotely. Performance tests may also be executed.

Tools installed on the local development system are not always the same as those installed on the CI system. In some organisations, static analysis tools on the CI system enable statistics for a time period to be collected when standalone software does not. This helps to identify the root causes of defects and can act as an input for the retrospective.

As seen in [Figure 1.13](#), depending on the technologies deployed, some organisations may execute dynamic analysis. For example, the way memory is released may be useful in C or C++, but not in Java. Some quality assurance actions, such as executing load tests or extracting documentation from the code, can be performed during the CI process. Reports may then be created by the CI and placed on an HTML page so that the team can consult them.

When code commits are successful within CI, developers have more confidence in their code and can quickly proceed to develop new features. The team can then move rapidly to a potentially shippable product and plan deployment in production. If the CI reports show errors, developers analyse and fix the defects, or add them to the backlog to be corrected at the beginning of the next iteration.

Risks to be mitigated with a continuous integration environment include:

- Continuous integration tools have to be introduced and maintained. For example, deploying a tool to support BDD is a project in itself. When introducing new tools, they must be compared with other competitor tools. Once the choice of tool is made, a proof of concept must be done and people, such as testers and developers, have to be trained in its use.
- The continuous integration process itself must be set up and established. This is a kind of operations process. It may be done on a simple PC, but updates have to be installed, so tests on the CI itself have to be undergone. For installation purposes, how to install any supporting tools must be studied and the time taken to do this recognised and accounted for.
- Test automation requires additional resources and can be complex to establish. Data sets have to be created, people trained to use the tools and time must be spent to create, for example, a keyword test

framework. Also, the test results analysis can be time-consuming if there are too many test scripts.

- Well-specified test cases are required to achieve automated testing advantages. This will increase the time needed for testing tasks.
- Teams sometimes over-rely on unit tests to the exclusion of system and acceptance tests. The test strategy must plan how upper level tests, such as system or end-to-end tests, will be executed.

## 1.2.5 Release and iteration planning

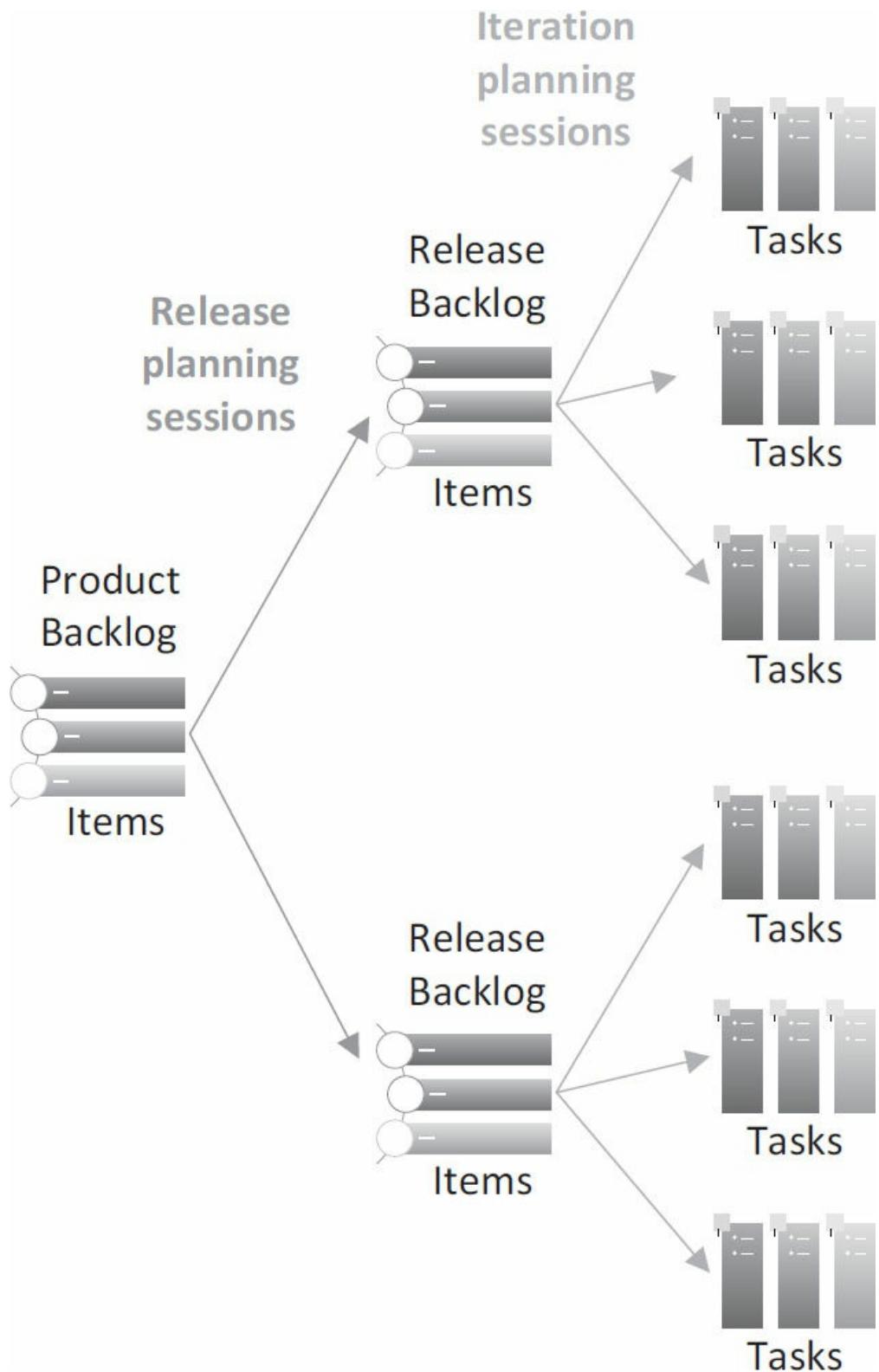
Release planning, as the name suggests, comes at the start of work on a new release, while iteration planning occurs at the start of each iteration within a release. Testers have a role in both. The following subsections explain both types of planning and the tester's role in each.

### Release planning

Several releases can be needed to develop a complete product. All the product features are listed in the product backlog. A release backlog is a subset of the product backlog (see [Figure 1.14](#)). A release is split into iterations, which will each deliver an increment containing a set of PBIs.

---

### Figure 1.14 Release and iteration planning



During release planning meetings, the team refines the release backlog progressively in order to have a list of items to be implemented. The items

must be sufficiently refined to be capable of being roughly estimated and dispatched in iterations providing value-added increments. Testers participate when the user's needs need to be refined. Testers bring a quality-oriented mindset to this activity and may propose product risk analysis on the items, suggest acceptance criteria and create rough estimates for testing effort.

The release planning meetings may begin sometime before the team begin to implement the items. This is because defining functions from the user perspective can take time until there are enough features defined for the early iterations. Also, the core architecture must be identified. Core functions, for example a framework, should be developed before the first iterations. Otherwise, the team will encounter a lot of changes, or people will be stopped because they will be forced to wait for structural components on which they will base their developments. Also during the first release planning phase, environment needs must be specified and the test strategy has to be defined. Testers will also assist with these activities.

The product owner establishes and prioritises the items for the release, in collaboration with the team, taking into account quality risks and user priorities. Release plans are high level because we have lists of features or strategies as a result, but we do not know exactly how to implement them.

The test approach is derived from the product backlog and its prioritisation. The test plan can span several iterations. For example, if the graphical user interface (GUI) is modified in each iteration to allow new features, regression tests must also be run against the GUI in each of these iterations. Also, the test approach can identify when end-to-end or performance tests can be executed without mocks.

## **Iteration planning**

The iteration planning meeting is held with both the team members and the product owner. The prioritised and refined released backlog items are the

basis of this meeting.

Iteration planning exists to identify the tasks needed to implement the backlog items and to assign them to team members. The tests needed as acceptance criteria become tasks to be performed as well as the functions to be coded and all other actions to be done. Iteration plans are low level, and stories, which at the release level may be estimated using story points, are broken down into tasks, which are generally estimated in hours.

During the iteration planning meeting, the team first defines the content of the iteration. For that, it elaborates the user stories, performs a risk analysis for the user stories and estimates the work needed for each user story. If a user story is too vague or insufficiently refined, the team can refuse it and use the next user story based on priority. Once the content is defined, the stories are broken into assigned tasks.

The number of stories selected is based on the established team velocity and the estimated size of the selected user stories. Testers participate in the following activities:

- Requirement management activities on user stories, as explained previously.
- Test planning activities, such as updating the test strategy and defining the necessary test types and levels; also, for regression testing, estimating testing effort for all testing tasks.
- Test design activities, such as creating acceptance tests for the user stories in conjunction with the team members and product owner.
- Breaking down user stories into tasks (particularly testing tasks).

## **Responding to change**

The release backlog must contain sufficient items for the first iterations to occur. It is recommended to have at least enough items for two iterations, but

change can occur in different ways as a result of internal and external factors.

- Internal factors:
  - Delivery capabilities, velocity and technical issues.
  - During an iteration, a feature may be more complex to implement than it first appeared.
  - Tests can take more time than estimated.
- External factors:
  - Discovery of new markets and opportunities, new competitors or business threats that may change release objectives and/or target dates.

So, the level of planning must be adapted. At the iteration level, more will be known about the impending stories than would have been the case when initial release planning occurred. The testware created by the testers must be both necessary and sufficient. Testers must see the big picture of the release to be able to adapt and plan its tests and environments.

When planning a release and the following iterations, we only have a clear vision of what can be done in the immediate future. For the subsequent steps, we have potential milestones and release dates.

## **Summary**

There are a number of different Agile development models that are used by software teams. We have looked at three of them, Scrum, Extreme Programming (XP) and Kanban. Despite their differences, all of them share some common approaches to development. These include the creation of user stories to drive the development process, the use of iteration and release planning, retrospectives as a means to achieve process improvement, and continuous integration as a mechanism to improve quality and provide

working software to customers.

## Test your knowledge

The answers to these questions can be found in the Appendix.

### Agile Tester Sample Questions – **Chapter 1 – Section 1.2**

#### **Question 1 (FA-1.2.1) K1**

Which of the following statements is true about Scrum, Kanban and XP in an Agile team?

#### **Answer set:**

- A. XP (Experienced Processes) is an organisational Agile framework
- B. Scrum and XP can be used together by a team
- C. Kanban is only used by Agile teams with more than 30 team members
- D. Scrum is mainly used by software maintenance teams

#### **Question 2 (FA-1.2.2) K3**

An Agile team has been created to develop a system of virtual keys for hotel check-in. Customers will enter a code to gain entry to their room. As this is a prototype, different versions of the software will be created. The team is exclusively made up of developers. There are only technical relations with the customer. The backlog, maintained by a project manager and an engagement manager, is not really maintained and is littered with tasks. During a retrospective, the team wonders if restarting from scratch would be a good idea. For that they need to change their way of working.

Based only on the given information, which of the following would MOST

LIKELY be a good way of working?

**Answer set:**

- A. Each developer executes exploratory tests on parts of prototypes and derives user stories from them
- B. Hire a tester to perform a product risk analysis with the customer
- C. Clean the product backlog and reprioritise items to match with the existing developed product
- D. Organise workshops with developers, customers and the product owner to redefine examples and how the feature must work

### **Question 3 (FA-1.2.3) K2**

Which of the following best represents how a retrospective is used as a mechanism for process improvement in Agile projects?

**Answer set:**

- A. The retrospective allows the team members to air grievances and leave in a good mood
- B. Improvements to processes and people occur when product improvements are identified daily
- C. Objective metrics and facts presented during a retrospective allow the team to think about improvements
- D. In the retrospective, all test cases are reviewed in order to update them with changes for the next iterations

### **Question 4 (FA-1.2.4) K2**

Which of the following statements is true within an Agile team?

**Answer set:**

- A. A continuous integration meeting helps to refine and integrate new stories in the product backlog
- B. Metrics and reports from the continuous integration environment provide efficient feedback to the team
- C. Continuous integration is an iterative test approach used for ‘big bang’ testing
- D. Continuous integration systems are used by the developer to check their code before they release it to the configuration management system

### Question 5 (FA-1.2.5) K1

Which of the following statements is true within an Agile team?

**Answer set:**

- A. Release planning explains accurately the prerequisites for releasing an increment
- B. Release planning meetings focus on acceptance criteria while iteration planning meetings focus on regression tests
- C. Release plans are high level while iteration plans are low level
- D. Release plans are low level while iteration plans are high level

## REFERENCES

Agile Manifesto (2001) Agile Manifesto. Available at [www.agilemanifesto.org](http://www.agilemanifesto.org) (accessed October 2016).

Beck, K. and Andres, C. (2005) *Extreme Programming Explained: Embrace change*. 2nd edn. Addison-Wesley Professional, Upper Saddle River, NJ, USA.

Crispin, L. and Gregory, J. (2009) *Agile Testing: A practical guide for testers and Agile teams*. Addison-Wesley Professional, Boston, MA, USA.

## FURTHER READING

Anderson, D. J. (2010) *Kanban: Successful evolutionary change for your technology business*. Blue Hole Press, Sequim, WA, USA.

Beck, K. (2000) *Extreme Programming Explained: Embrace change*. 1st edn. Addison-Wesley Professional, Boston, MA, USA.

Cohn, M. (2004) *User Stories Applied: For Agile software development*. Addison-Wesley Professional, Boston MA, USA.

Cohn, M. (2009) *Succeeding with Agile: Software development using Scrum*. Addison-Wesley Professional, Boston, MA, USA.

Martin, R. C. (2002) *Agile Software Development: Principles, patterns and practices*. Pearson, Harlow, UK.

Rubin, K. S. (2012) *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley Professional, Upper Saddle River, NJ, USA.

## WEBSITES

[www.agilealliance.org](http://www.agilealliance.org)

[www.scrumalliance.org](http://www.scrumalliance.org)

[www.mountaingoatsoftware.com](http://www.mountaingoatsoftware.com)

# **2 FUNDAMENTAL AGILE TESTING PRINCIPLES, PRACTICES AND PROCESSES**

by Marie Walsh, Jan Sabak and Bertrand Cornanguer

In [Chapter 1](#) we looked at the fundamentals of Agile software development, the Agile Manifesto and the various Agile approaches. In [Chapter 2](#) we will look at testing within an Agile development environment and how this differs from traditional software development approaches, and explore the principles, practices and processes used in an Agile environment. We will also explore the role and skills required of a tester in an Agile team, and how as a tester you can transfer the skills you have acquired as a traditional tester into that of an Agile tester.

## **2.1 THE DIFFERENCES BETWEEN TESTING IN TRADITIONAL AND AGILE APPROACHES**

The learning objectives for this section are:

FA-2.1.1 (K2) Describe the differences between testing activities in Agile projects and non-Agile

projects

FA-2.1.2 (K2) Describe how development and testing activities are integrated in Agile projects

FA-2.1.3 (K2) Describe the role of independent testing in Agile projects

Now that we have an understanding of Agile development approaches from [Chapter 1](#) of this book, we are going to look at how we as testers are able to successfully transition from testing in a traditional lifecycle model (for example, a sequential model, such as the V-model, or an iterative model, such as Rational Unified Process (RUP)) to testing in an Agile lifecycle.

To be able to transition into an Agile testing role successfully, and to be effective and efficient within the team, requires us first to understand the differences between both lifecycle models, including the differences in development and testing activities, project work products, test levels, exit and entry criteria, use of tools and how independent testing can be effectively utilised.

## 2.1.1 Testing and development activities

In [Chapter 1](#), we looked at how Agile development approaches have short iterations with the concept of having working software that delivers features of value to the business at the end of each iteration. To enable this to occur, as testers we need to be able to identify the differences between testing in a traditional environment and testing in an Agile environment and be able to adapt our approach to work within an Agile team. So, what are these differences?

The primary differences include:

- The involvement of testers throughout the entire release.
- Continual testing throughout an iteration.
- Coaching, collaboration and pairing.

- Flexibility to accommodate change.

One of the first differences you will notice is that testing is involved throughout the entire release in an Agile environment. From initial involvement in release planning activities involving all members of the Agile team, including business stakeholders, developers and testers, as a tester you would provide input to user stories and task risk analysis and sizing (estimation) activities. Using risk-based testing techniques, you can assist in driving the high-level risk analysis for the release, which will feed into the establishment of each iteration based on the risk level, sizing of the user stories and the build approach. Refer to [Section 3.2](#) for further information on risk-based testing in Agile projects.

Testers are then involved in iteration planning, working closely with business stakeholders and developers to define the acceptance criteria of each user story before development activities commence (further details of acceptance criteria can be found in [Section 3.3](#)). At this point, the quality risks associated with the iteration are analysed. This analysis can assist in determining when features in the iteration are developed, and the priority and depth of testing required for each feature. To succeed in Agile development, the team must collaborate and continually work together on build, integration and testing activities; more so than how you work together in traditional development teams.

The next major difference in an Agile development approach is the concept of continual testing throughout the iteration. In a traditional development environment, developers perform unit testing during development activities, and formal integration, system and acceptance testing does not commence until all code development of the project is completed and a full software solution is delivered into a test environment. Traditional development approaches also have the concept of test levels with associated entry and exit criteria (as described in Section 5 of the Agile Foundation syllabus), which

are used as gates to be able to proceed to the next test level. Within an Agile team, these gates are no longer formalised; instead, test levels overlap during the iteration, with testing commencing as soon as development does. Developers continue to perform unit testing during development of a feature, with testers now taking regular software builds into the test environment (generally from a continuous integration [CI] system) and verifying stories as they are developed. These builds can be hourly, daily or on-demand – whenever the team is ready to take a new build. However, to align with Agile approaches, they should be taken as frequently as possible, reducing the lag between development and testing. One of the main goals of Agile is to provide early and regular feedback on the quality of the software as it is built. Testing is not considered a final activity as it is in a traditional approach; there is constant parallelism and overlap of testing activities throughout the iteration. Developers, testers and business stakeholders conduct testing activities throughout the iteration, with constant early feedback being provided to allow quick course correction.

As with traditional approaches, quality is everyone's responsibility. In an Agile team, testers, developers and business stakeholders all have a role in testing. To enable all team members to collectively take ownership of the product's quality, as a tester you may also take on the role of a testing and quality coach, sharing testing techniques and knowledge with all team members. This assists in developing the capability of the team, and promoting the core Agile principles.

Throughout the iteration, testers and developers work closely together and, in some teams, pairing of two testers, or a tester and a developer is the preferred approach. Having paired developers and testers enables stronger collaboration during the development and testing of a feature. Testers are able to provide input to the developer on what they would be looking at testing for the feature, which enables the developer to consider how the code for the

feature should handle these scenarios.

### **Author example**

While I was working in a banking integration services team in the early days of Agile (in 2002), I was paired with a developer. Both of us were new to Agile and to the concept of testers and developers working together at the same time. We started off badly! We fell into traditional working behaviours and were working independently of each other. We quickly realised that it was not working. We then started reviewing the features together, determining the acceptance criteria and discussing how they would be tested. The developer then built these scenarios into unit tests. While he was coding and writing the unit tests, I would peer review these with him. While I was developing my system integration tests and automating them, he would peer review these with me. All our code, unit tests and system integration tests were checked into the same source code repository and we had a CI server that would perform automated builds upon code check-in, which gave us instantaneous feedback of the build quality. Working closely with the developer improved the overall quality of the code at the same time as delivering working software a lot quicker to the business.

As the years have passed, with the various development teams I have joined, in all cases where I have paired with a developer, I have had the opportunity to learn more about the code and how it is constructed. It has assisted me in being able to identify technical risks a lot easier, at the same time allowing me to improve and grow as a tester and gain an understanding of development activities. At the same time, the developers I have worked with have also had the chance to see the product through the eyes of a tester, and it has allowed them to learn new techniques on how to test their code.

Following on from this, the next difference we experience between traditional and Agile approaches is the ability to accommodate changing requirements. Agile methods enable a quick response to change because the business stakeholders see the system as it is being developed, since they are involved in testing activities, either formally or via experimentation with the features as they are developed. In traditional development approaches, the requirements are locked down or frozen well before development and testing commences, which does not provide the flexibility to adapt and change course during development. In Agile, as the business stakeholders are involved all the way through, they see the product as it is evolving and are

able to work with the Agile team to make corrections and amendments as they see the product taking shape.

One of the major benefits of an Agile development approach is how a feature or user story moves through the lifecycle. A feature is not considered ‘done’ until it has been integrated and tested with the system, which reduces the risk of technical debt.

Technical debt is a metaphor that was first used by Ward Cunningham (Cunningham, 2009) and is regularly used by Agile teams to describe any productivity inefficiencies or losses (debt) accumulated on the source code or architecture when delivering code that is not quite right for its intended purpose.

As with traditional development approaches, technical debt (for example, defects) can be accepted into the next phase or iteration of development. Technical debt can be more than just defects from the previous iteration, and may involve situations where features that are built may meet the current requirements, but have been deemed to not be scalable. An example of this is the use of a less-efficient data structure to hold an inventory of items that meet the current business requirements. This is not strictly a defect, at least not a functional one, but the implementation will not scale as the number of items in the inventory grows. It is good practice to address the technical debt from the previous iteration as part of the backlog for the next iteration prior to moving on to new features ('fix bugs first'), but this can come at a cost to the ability to deliver new features. Most experienced Agile teams ensure that technical debt is kept to a minimum throughout each iteration. If teams are experiencing a large volume of technical debt, the introduction of a periodic stabilisation or hardening iteration is recommended. This allows teams the opportunity to concentrate on stabilising the product, and reducing the technical debt being carried forward. Throughout this process, the team needs to remain committed to the goal: deliver working software that delivers features of value to the business at the end of each iteration.

As the product is being constantly worked on and built up over numerous iterations, it is imperative that test automation at all levels of testing be part of the solution to reduce the risk of regression (refer to [Section 2.2](#) for details on managing regression risk), especially with changing requirements during an iteration. The use of test automation allows the team to manage the amount of test effort associated with change, but at the same time a balance needs to be met within the team to ensure the level of change does not exceed the team's ability to deal with the risks associated with these changes.

Within an Agile team, testers spend more time creating, executing, monitoring and maintaining automated test cases and results, therefore manual test approaches within an Agile team have adapted to be more focused on experience-based and defect-based techniques, with a higher emphasis on exploratory testing and more lightweight work product documentation.

## **2.1.2 Project work products**

Within traditional development teams, project work products consist of vast amounts of very detailed documentation, from thick business requirements specifications, to heavy test strategy and test plan documents. Agile development teams favour the Agile value of ‘working software over comprehensive documentation’. This does not mean that no documentation is produced – it means that only meaningful documentation that provides value to the team is produced – this is also known as ‘just enough’ documentation. A balance must be struck between increasing efficiency by reducing documentation and providing sufficient documentation to support business, testing, development and maintenance activities.

Agile teams also need to be able to quickly adapt to changing requirements. Within traditional development teams, the detailed business requirements documents are produced at the start of the project and are locked down before

development commences. The ISTQB® glossary refers to these locked down documents as a frozen test basis. This freezing of the test basis restricts the team's ability to quickly react to change. Within Agile teams, requirements (user stories) are reviewed at the start of each iteration, and are amended and updated as the product starts to take shape and the business stakeholders are able to see their earlier requirements becoming a tangible product that they can use. This practical feedback loop enables them to make modifications to future requirements before the build for the next iteration commences.

Project work products that are of interest to Agile testers are:

- business-oriented work products;
- development work products;
- test work products.

These are explained in the following three subsections.

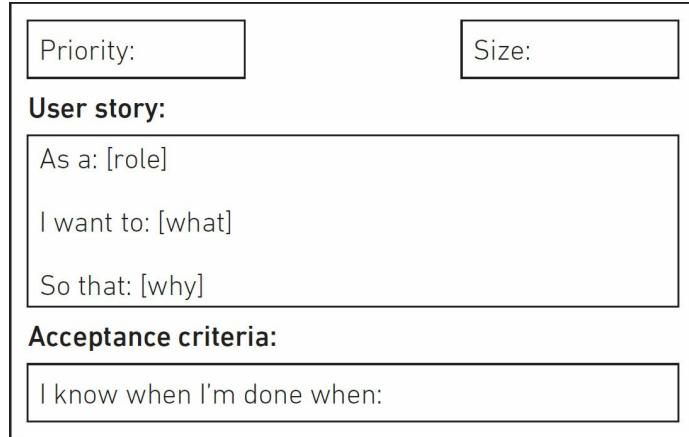
### **Business-oriented work products**

Business-oriented work products describe what is needed (for example, requirements specifications) and how to use it (for example, user documentation).

Within Agile teams, user stories are used to define the business requirements. User stories provide a lightweight approach to managing requirements, providing a short statement of a single function. User stories (see [Figure 2.1](#)) are captured either on an index card or in a tool. The story acceptance criteria are then recorded, either at the bottom of the story or on the back of the index card. All user stories must contain acceptance criteria before they can be considered for inclusion in an iteration.

---

#### **Figure 2.1 Example of a story card**



Collections of related user stories or large and complex stories are known as epics. An epic is essentially a large user story that can be broken down into a number of smaller stories, which may take several iterations to complete. Each epic and its user stories should have associated acceptance criteria.

## **Example of an epic**

**Epic:** Increase self-service functionality on our internet-banking portal

**User story 1:** As a customer, I want to access my historical statements so that I can retrieve statements over the past three years.

**User story 2:** As a customer, I want to transfer money to an external account so that I can increase my personal banking capabilities.

**User story 3:** As a customer, I want to set up regular automatic funds transfers so that transfers will automatically occur on set schedules.

**User story 4:** As a customer ...

**User story 5:** As a customer ...

...

**User story n:** As a customer ...

## **Development work products**

Development work products usually consist of the following types:

- Work products that describe how the system is built; for example, database entity-relationship diagrams and software architecture diagrams (such as Unified Modeling Language (UML) diagrams).
- Work products that are used to actually implement the system; for example, code.
- Work products that evaluate individual pieces of code; for example, automated tests.

There have been significant advancements in development tools, where developers are able to either convert architectural diagrams into code via code generators or convert code into architectural diagrams and system documentation. Tools that convert code into architectural diagrams assist developers to understand the design and the code dependencies and to visualise the sequence of calls that happen in key sections of the application. These insights enable them to design new functionality easily and to verify the interactions with existing code. These diagrams are often used as the technical documentation in describing how the system is built.

In addition to code and unit tests, some developers on Agile teams subscribe to test-driven development (TDD) techniques, where test cases are created prior to the code being developed. These tests are executed to verify the new code works as expected. TDD is a way that developers can think through the requirements and the design of the code before actually writing any code. TDD can also be considered a form of executable low-level design specifications. Tests are then added, which incrementally improve the design. TDD tests are another form of documenting how the system works. Further details on TDD can be found in [Section 3.1](#).

## **Test work products**

Test work products usually consist of the following:

- Work products that describe how the system is tested; for example,
- test strategies and plans.
- Work products that actually test the system; for example, manual and automated tests.
- Work products that present test results; for example, test dashboards.

The majority of traditional development environments consist of a large amount of detailed test documentation that has to be maintained throughout the system's lifetime. These documents consist of detailed test strategies, test plans, manual test cases, test reports and defect reports. Agile teams work in short sprints or iterations, which consist of tasks where the system is designed, coded and tested. Agile testers need to be able to balance the level of detail contained in their work products with the ability to keep up with the fast pace of Agile development. This is not to say that test documentation should not be done. Testers need to ask themselves what purpose the documentation will serve, and develop the required level of documentation to satisfy the team's objectives.

Test strategies in Agile teams have evolved into a whole-team test strategy, defining the quality aspects for both functional and non-functional requirements. The test strategy can take many different forms: the traditional document style format, use of a wiki, sticky notes or on a whiteboard. There are no rules; it is what works best for your team, and to which the whole team contributes. Test strategies should contain information that is relevant to the whole team, so there is a shared understanding of how testing is to be performed on the project. The test strategy should:

- Describe the approach to verification and validation activities.
- Define quality success criteria in terms of measures and metrics for all test activities.
- Provide stakeholders with an understanding of how and what the test

approach will be for the whole team.

It is a common practice in Agile teams that more detail is put into the test strategy, and then shorter (generally single page) test plans are produced during iteration planning activities. Many Agile teams develop their test plans on flipcharts or whiteboards or in mind map tools – whichever method, the whole team contributes and agrees about what is sufficient to ensure quality for that iteration. This reinforces that quality is everyone's responsibility. The purpose of a test plan in Agile teams is to outline what you are going to test in that iteration, how you are going to test it and how you confirm you meet the acceptance criteria.

Another difference to traditional approaches is that Agile teams rely heavily on test automation (as described in [Section 2.2](#)). As a result of the shorter timeframes in Agile development and the constantly evolving application code, the team relies heavily on test automation to continuously verify the software as changes are being made. Automation of tests at all levels (for example, unit, integration and acceptance) and for all test types (for example, functional and non-functional, such as performance and security testing) enables software changes to be rapidly verified and then potentially deployed. Test automation scripts form a critical role in test work products as they describe how the system has been tested.

As with traditional approaches, testers on Agile teams also perform manual testing. However, on account of the heavy emphasis on automation of repetitive testing activities, manual testing focuses more on exploratory testing. The work products generally produced during manual testing include:

- test charters;
- checklists;
- mind map diagrams;
- exploratory test logs.

Refer to [Section 3.3](#) for details on exploratory testing.

Agile teams rely on information being highly visible to all interested parties, from the task board through to team status dashboards. Dynamically updated status dashboards are used rather than traditional detailed test reports.

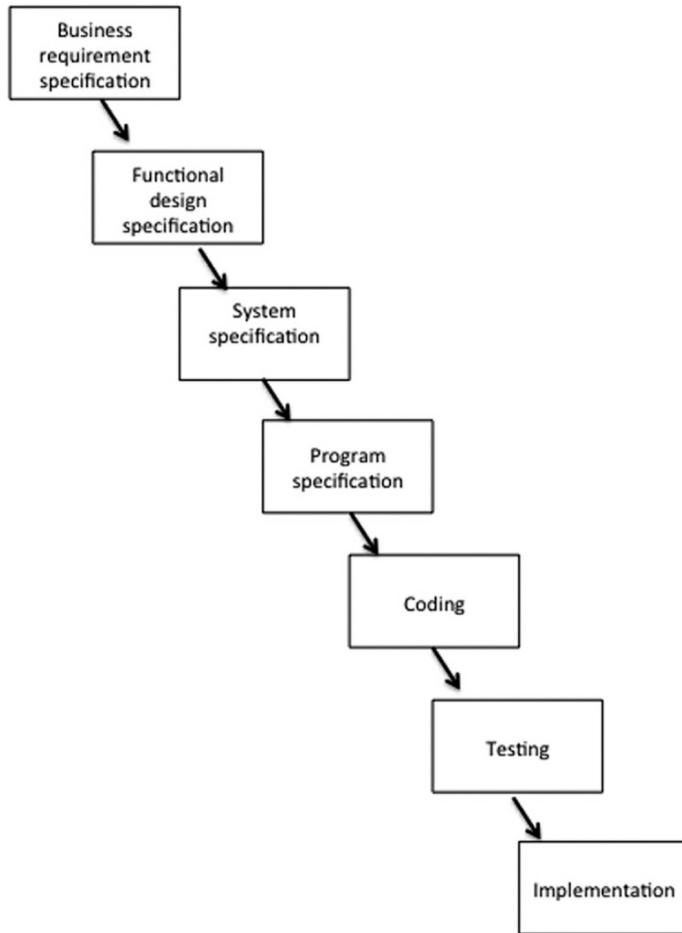
As with all development approaches, understanding the nature of the software/application that is being developed is critical. In cases of safety-critical systems, systems bound by regulatory requirements or highly complex systems, more rigour may be required around the level of detail contained in all the work products produced by the Agile development team. In some cases, where required to satisfy auditors or regulatory bodies, user stories, acceptance criteria, system designs, test plans and test reports may have to be transformed into more formalised documentation, and show full traceability between all work items to ensure the team meets the necessary conformance requirements.

### 2.1.3 Test levels

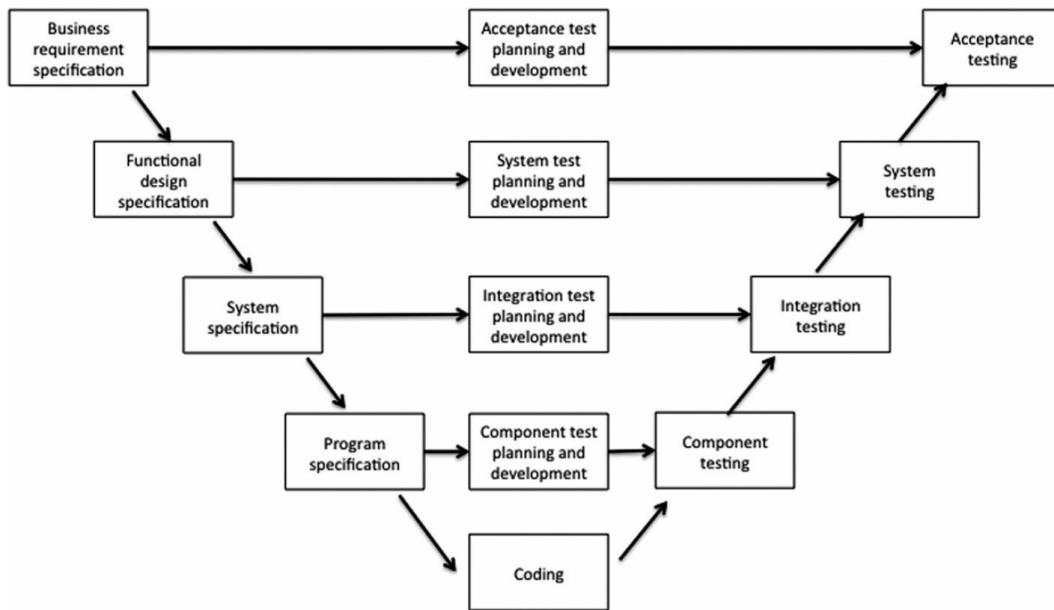
Within a traditional development approach, there are set test levels, each having defined entry and exit criteria to allow movement between the levels. In the Certified Tester Foundation Level (CTFL) syllabus, we became familiar with the traditional waterfall approach to software development (see [Figure 2.2](#)). We also learnt to recognise the test levels that fitted with it in the V-model (see [Figure 2.3](#)).

---

**Figure 2.2 Waterfall model**



**Figure 2.3 V-model for software development**

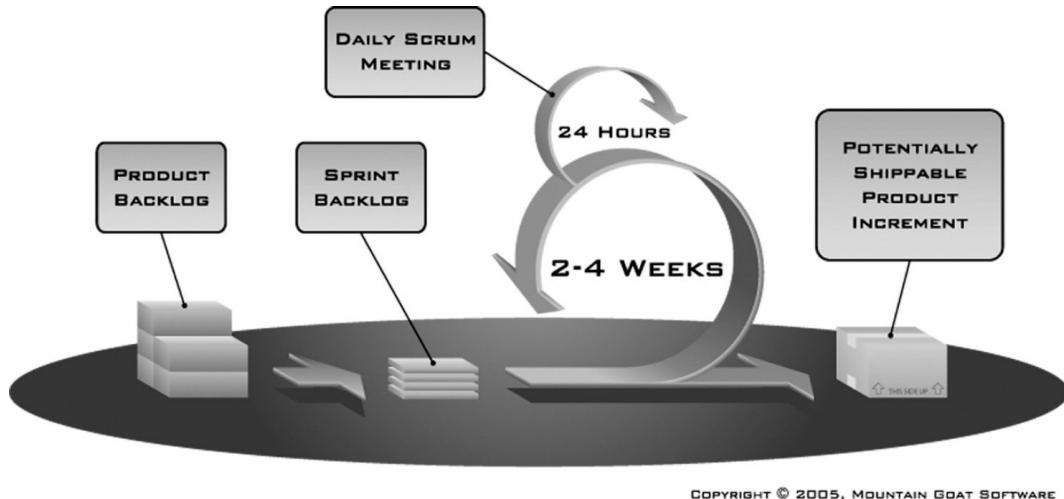


With the V-model, the test levels are clearly defined, with each containing entry and exit criteria that must be met before being able to move to the next test level (refer to [Chapter 5](#) of the CTFL syllabus).

An Agile approach to software development, as described in detail in [Chapter 1](#), involves shorter design, development and test timeframes. These are generally two to four weeks in length, and the end result of the iteration or sprint should be ready to be deployed to production, as demonstrated in [Figure 2.4](#).

---

**Figure 2.4 Agile Scrum approach**

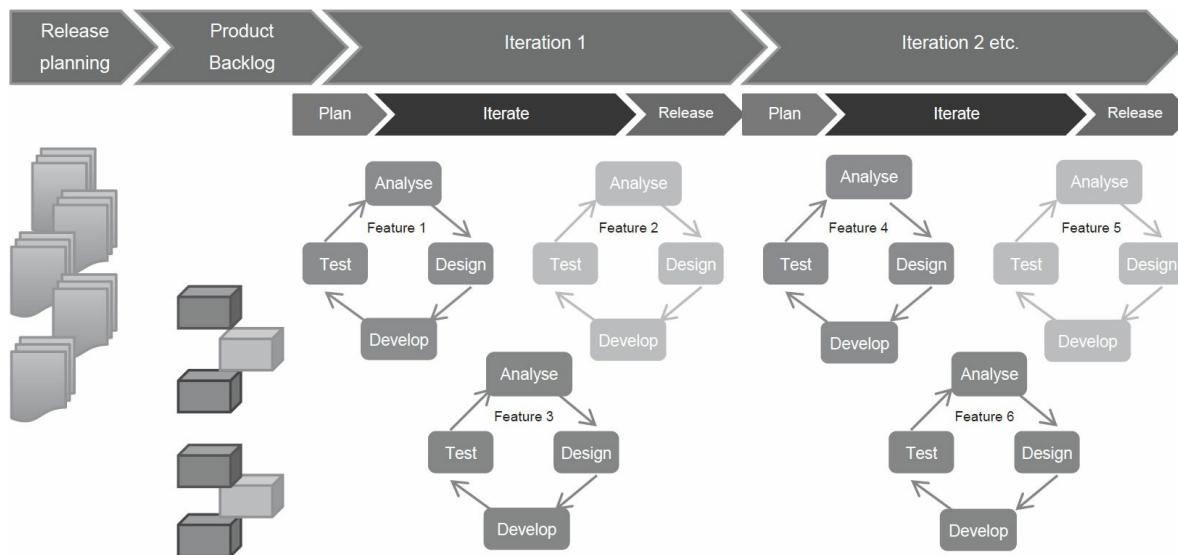


**Note:** Diagram courtesy of Mountain Goat Software and reproduced with permission (<https://mountaingoatsoftware.com/agile/scrum/images>). Distributed under the Creative Commons Attribution 2.5 License (<https://creativecommons.org/licenses/by/2.5/>).

One of the more significant differences between traditional and Agile development models is the approach to testing. Testing in Agile development involves team collaboration in which testers are involved up front in the release planning and iteration planning activities – providing input into user stories, acceptance criteria and estimations. Once an iteration kicks off,

testers are involved immediately, testing the user stories as the developers code them, as demonstrated in [Figure 2.5](#).

**Figure 2.5 Agile development and test**



There is no longer a concept of entry and exit criteria as a form of staging, since most activities are done in parallel, with test levels and activities overlapping each other. Generally, a developer will still conduct unit testing at the same time as writing the code, as with traditional development approaches. However, this user story is then made available to testers immediately, allowing them to perform the necessary test activities to verify the user story meets the defined acceptance criteria, and validate the user story is fit for use.

Feature verification confirms that the software works as per the user story acceptance criteria. It can be conducted by either a developer or a tester, and is usually automated, allowing it to be continually run throughout the iteration, and later iterations, providing assurance that the story continues to meet its acceptance criteria as the software iterates through the release.

Feature validation activities are used to confirm that the software is fit for

use. They are generally manual tests with involvement from developers, testers and business stakeholders working collaboratively. Feature validation enables business stakeholders to see the progress being made and creates an avenue for them to provide immediate feedback to the team.

As demonstrated in [Figure 2.6](#), features are developed and then built upon in each iteration. This necessitates the need to perform constant regression testing throughout the iteration, and in all subsequent iterations. Agile teams use a CI framework running unit and feature verification tests continually (with each build, at the end of each day etc.) containing all automated tests from the current iteration and previous iterations to reduce the risk that regression faults have been introduced in the evolving software code.

---

**Figure 2.6 Features delivered per iteration**

Iteration 1	Iteration 2	Iteration 3
<p>Feature 3</p> <p>Feature 1</p> <p>Feature 2</p>	<p>Feature 7</p> <p>Feature 5</p> <p>Feature 3</p> <p>Feature 1</p> <p>Feature 6</p> <p>Feature 4</p> <p>Feature 2</p>	<p>Feature 9</p> <p>Feature 7</p> <p>Feature 5</p> <p>Feature 3</p> <p>Feature 10</p> <p>Feature 8</p> <p>Feature 6</p> <p>Feature 4</p> <p>Feature 2</p>

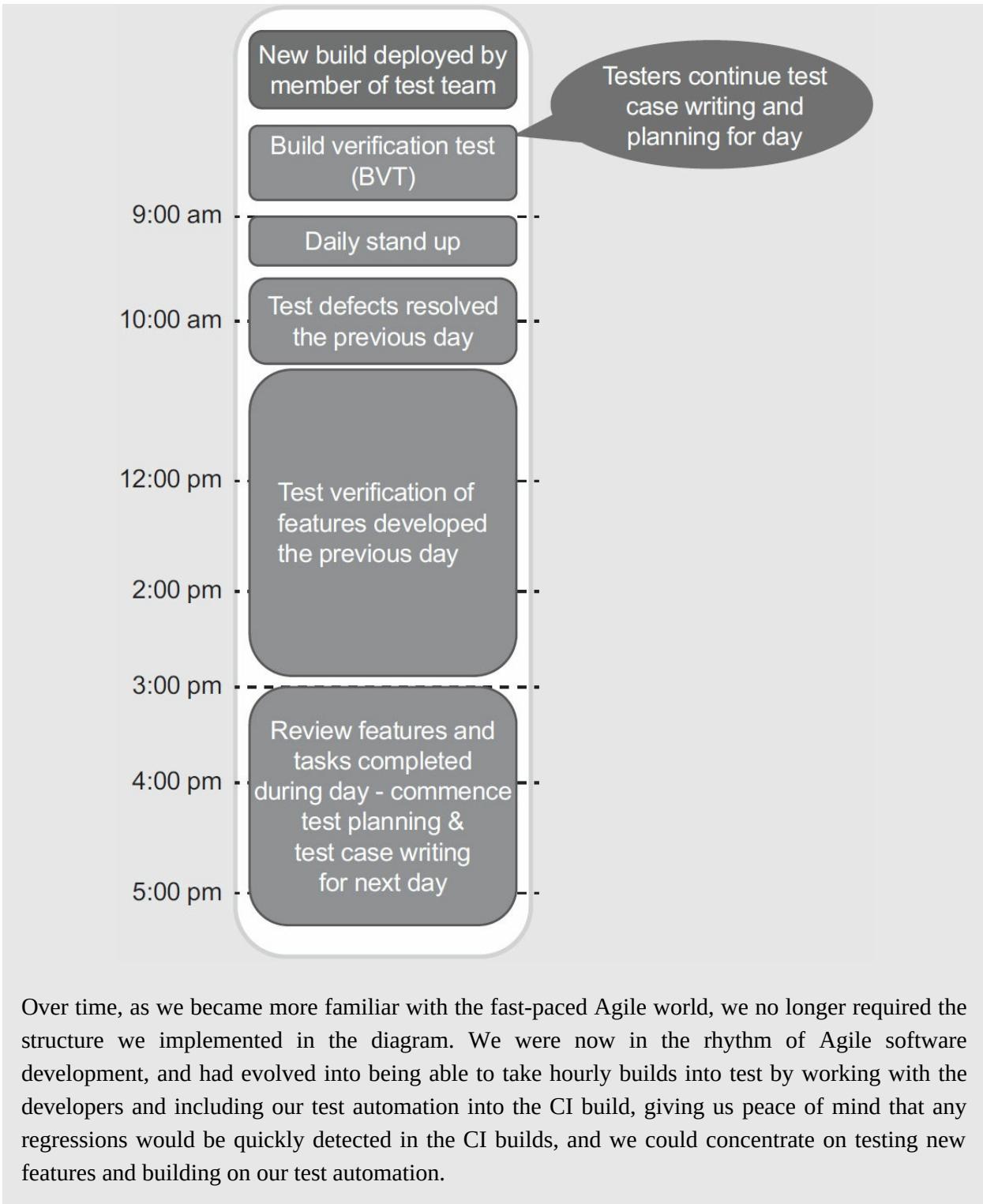
Test activities conducted throughout the iteration are dependent on the features being developed. Within an iteration there may be a requirement for non-functional testing such as performance testing, reliability testing, security testing or usability testing. This is generally decided during iteration planning

activities, and the team work together to identify the best time within the iteration to perform these test activities, as they may require more than a single feature to be completed prior to being executed.

Similar to traditional development approaches, an Agile team may decide to include a specific user acceptance testing activity, operational acceptance testing activity, regulation acceptance testing or contract acceptance testing activities (refer to definition in Foundation Level syllabus). These can occur at the end of an iteration or after several iterations, and are done in addition to the ongoing business stakeholder validation activities occurring throughout an iteration. This allows the business stakeholders to validate the whole system (as built) rather than specific features they would have focused on during the iteration.

### **Author example**

In one of my earlier Agile projects we split the project team into five pods of three developers and one tester per pod. We were all working on the same product, but it allowed us to work on separate areas of the product. As testers relatively new to Agile, we were lost on how to manage our day as we now had features being delivered to test daily, and needed to keep up with the development team. We no longer had the traditional test levels, exit and entry criteria, and also had a much shorter timeframe to complete testing activities. After a few days, we realised that we were no longer working as a test team, and ended up impacting each other. After yet another instance of someone deploying a build into test without the other members of the test team knowing about it, we came up with a plan on how to manage our day, and what daily test activities we would need to perform to be able to get through the iteration. This approach may seem very structured to some. However, it allowed us the ability to transition from traditional test approaches into the fast-paced world of Agile, and to maintain our test team communication and collaboration, while also keeping up with the daily features being delivered to test.



## 2.1.4 Testing and configuration management

Traditionally, developers have been the primary users of configuration

management tools for code source control and management on a project. Within Agile teams, with test automation being more prevalent, it has become necessary for testers to also utilise configuration management tools to ensure correct versioning and control of their test automation code. This also allows testers to ensure that their version of automated tests correctly aligns with the version of the code, which is critical given the amount of code churn on Agile projects.

Having the system and acceptance test automation code in the same configuration management tool as the software source code and unit tests enables the team to include them in the CI framework, providing immediate feedback on code quality. Most teams partition their CI framework to ensure that only the software build and unit tests are run with every code check-in. This is due to the length of time that the automated system and acceptance tests can take to run, which could potentially cause a large backlog in builds. A full CI build and test run is generally scheduled throughout the day or every few days (depending on the number of tests and length of time to run). It is recommended practice not to go too long without running the full suite, as this can delay finding defects that may have been introduced, as unit testing alone has limited defect detection effectiveness.

The benefit of regularly running the full suite of automation is the immediate feedback that the build is functioning, stable and installable. Any failing tests are generally investigated and a code fix checked in prior to the next build, in order to ensure reduction in technical debt and enable the team to continue building on a stable code base. These types of defects are normally found too late in the release in traditional approaches, therefore having the potential of a larger impact due to coding on top of defective software.

CI tools provide reports on each build, allowing the team to quickly identify any problems, down to the component or line of code that is failing (for unit tests), or the functional area failing with system and acceptance tests. This

provides an additional advantage for testers, as the CI framework has the software builds available to be deployed, either manually or via an automated process, into test environments. Testers are able to either ‘pull’ (download) a build and manually deploy it in their environment, or ‘push’ (publish) a build that automatically deploys in their environment. Automated build deployments generally require additional software to be integrated with the CI software. These methods allow testers to take new software builds into their test environment whenever they need to during a sprint or iteration. It is recommended that testers on Agile teams take new builds into test at least once a day, but preferably more frequently. The frequency of builds needs to be balanced between being frequent enough to provide quick feedback, but not too frequent to create an overhead or be too disruptive. Utilising a CI framework and the ability to take frequent builds into test reduces the regression risk associated with constantly changing code, which is discussed in more detail in [Section 2.2](#).

### **2.1.5 Organisational options for independent testing**

An Agile team includes three roles: the product owner, the Scrum master or iteration manager, and the development team. So, where does testing fit in? Within the development team! The development team consists of people with differing skillsets, including programmers, testers, designers and anyone else the team requires in the product development (such as technical writers etc.). There was originally a concept that all members of the development team should be able to perform any role within the team. However, as Agile has evolved over the years, there has been the realisation that people with specific skillsets should perform specific roles in the team, such as programmers designing and coding, and testers testing, as they are often more effective in finding defects. At the same time, all roles are learning from each other, and contributing to each other’s tasks, embracing team collaboration.

When it comes to testing, there are three main ways that testers are integrated into the Agile team. As with every implementation of a development methodology or approach, the way testers are integrated into the team depends on the team's needs. There is no right or wrong way; however, there are benefits and disadvantages of each implementation.

The first method is to have developers building automated tests with testers embedded in the team performing testing tasks. The benefits and disadvantages of this approach are:

- Benefits
  - Testers can concentrate on the manual testing of user stories.
  - Testers are part of the team from the beginning, therefore they understand the applications being built.
  - Testers build a strong relationship with the team.
- Disadvantages
  - Testers can possibly lose their independence by being part of the team, without the support of other testers in the organisation.
  - Test automation is being built by developers and therefore may not contain the same level of verification or detail that it would contain if written by testers.

The second method is to have a fully independent and separate test team in which testers are assigned to the project on-demand, generally at the end of the sprint or iteration. The benefits and disadvantages of this approach are:

- Benefits
  - Testers can maintain their independence.
  - Testers can provide an objective and unbiased evaluation of the software.

- Disadvantages
  - Testers face time pressures to test features developed over the entire iteration in a very short period of time.
  - Testers do not have knowledge of the product, and lack understanding of the features in the product.
  - Testers have not had the opportunity to build a relationship with the business stakeholders and developers, which can lead to communication and interaction problems.
  - This method defeats the objective of Agile development.

The third method is to have an independent and separate test team in which testers are assigned to Agile teams on a long-term basis from the commencement of the Agile project. The benefits and disadvantages of this approach are:

- Benefits
  - Testers maintain their independence.
  - Testers gain a good knowledge of the product and understanding of the features in the product.
  - Testers build a strong relationship with the business stakeholders and developers.
  - Testers can leverage other testers outside the Agile team for tooling or specialised support.
  - The independent team can have specialist testers outside the Agile team working on activities such as:
    - ◆ automated test tool development;
    - ◆ performing non-functional testing, such as performance testing, security testing and so forth;

- ◆ supporting and managing test environments;
- ◆ developing or generating test data to support test activities;
- ◆ supporting multiple Agile teams by performing test activities that might not fit well within a sprint or iteration.

- Disadvantages
  - This approach works well within large organisations, but is not really suited for smaller organisations with limited resources.

Each of the methods identified above are valid, and choosing the right method depends on your team setup. The one thing they all have in common is that there is a requirement for independent testing on all Agile teams.

## **Summary**

In this section, we have discussed how a tester can successfully transfer from a traditional testing role into an Agile testing role, the differences between testing in both environments and the need to adapt our testing approach to be able to accommodate constantly changing requirements within an Agile team. We have also learnt: in Agile teams, testing is involved throughout the entire release; testing is continual throughout an iteration, and commences as soon as development does; test levels are not as clearly defined and often overlap in each iteration; and the benefits of pairing and coaching within the team, especially between developers and testers, to enable stronger collaboration during the development and testing of features.

In addition, we looked at the business-oriented, development and test work products that are of interest to Agile testers, also discussing the balance between increasing efficiency by reducing documentation while providing sufficient documentation to support business, testing, development and maintenance activities, and the need for testers to utilise configuration management tools to ensure correct versioning and control of their test

automation code.

We also looked at the organisational options for independent testing, identifying that there is a requirement for independent testing on all Agile teams, and choosing the right method depends on your team setup.

## **Test your knowledge**

The answers to these questions can be found in the Appendix.

### **Agile Tester Sample Questions – Chapter 2 – Section 2.1**

#### **Question 1 (FA-2.1.1) K2**

Which one of the following is the main difference between being a tester on an Agile team or being a tester on a traditional project?

#### **Answer set:**

- A. Testers create and execute automated tests
- B. Testers closely collaborate with developers and business stakeholders
- C. Testers commence test execution activities as soon as development commences
- D. Testers perform exploratory testing

#### **Question 2 (FA-2.1.2) K2**

Which TWO of the following statements are true on Agile projects?

#### **Answer set:**

- A. The Scrum master or iteration manager manages the product backlog and decides which stories, features and tasks will be worked on in

- the next iteration
- B. Business stakeholders, developers and testers collaborate to define user story acceptance criteria
  - C. Testers work with developers to write the unit tests and are responsible for unit testing on Agile projects
  - D. Testers and developers pair on Agile projects and testers provide input to the developer on what tests they would consider for the feature
  - E. Developers write all test automation scripts, allowing testers time to manually test all new features

### **Question 3 (FA-2.1.3) K2**

Which of the following statements about testing on Agile projects is true?

#### **Answer set:**

- A. All members of the Agile team should be able to perform any role within the team, including testing
- B. Testers on Agile projects are only ever brought in at the end of the iteration or sprint to perform independent testing activities
- C. Independent testers will find more defects than developers regardless of test level
- D. The way testers are integrated into the team depends on the team's needs

## **2.2 STATUS OF TESTING IN AGILE PROJECTS**

The learning objectives for this section are:

FA-2.2.1 (K2) Describe the tools and techniques used to communicate the status of testing in an Agile project, including test progress and product quality

FA-2.2.2 (K2) Describe the process of evolving tests across multiple iterations and explain why test automation is important to manage regression risk in Agile projects

Agile development approaches embrace change. They are designed with the principle that change is good rather than an obstacle. Change within an Agile project is a normal thing. There is an old saying that the only constant thing in a software project is change. So, as Agile projects constantly evolve, so too does the test status, test progress and product quality. All project stakeholders need to be constantly aware of test progress and product quality to enable them to make informed decisions accordingly. The use of tools enables teams to efficiently and effectively track and communicate the project's progress and quality.

With constant change comes the need for maintaining both manual and automated test cases to ensure they evolve with the product. This also facilitates the need for more comprehensive test automation to assist with managing regression risk, which is prevalent in a constantly changing product.

## **2.2.1 Communicating test status, progress and product quality**

Good Agile teams know the status of their work and update it often. Short iterations require effective and clear communication between team members. Each team member should be aware of the progress towards the iteration goal, as the responsibility of achieving that goal is that of the team as a whole, regardless of the roles of individual team members. Thus, Agile teams use various methods and tools to record test status, progress and product quality. Most of these tools are lightweight and automated where possible, so testers can put more effort into actually testing, instead of preparing tediously long reports.

The most common method for conveying status information is via the daily

stand-up meeting. Testers, as Agile team members, take part in stand-ups, including answering the same three basic questions asked of the other team members:

- What have you completed since the last meeting?
- What do you plan to complete by the next meeting?
- What is getting in your way?

These questions allow testers to share knowledge about testing and the status of testing tasks as well as information about the quality of the products being developed. Any blocked tests or major risks should be communicated here.

Another commonly used tool is the project task board holding tasks needed to complete (or ‘burn down’) the iteration backlog. The project task board can either be a physical board on the wall containing sticky notes or index cards in various colours to indicate a user story, task, defect and so forth or an electronic task board with automated workflows built in. A task board shows the status of all user stories and tasks included in the current iteration at any given point in time. An example of a task board used on one of the authors’ projects is demonstrated in [Figure 2.7](#).

The first column in [Figure 2.7](#) contains the iteration backlog; i.e. all user stories that are to be implemented during the iteration. This example also shows two technical stories. One is ‘Sprint Impediments’ that will contain any task that stops execution of other tasks. There are no impediments in this project at the moment. The other is ‘Bugfixing Previous’ that will contain tasks related to defects reported against user stories and features developed during previous iterations. These features are not present on this task board, but regression tests can reveal additional defects that have not yet been found.

The first row contains the sequence of steps needed to develop a feature. In this case, it is ‘New’, ‘In Progress’, ‘Review’, ‘Testing’, ‘Done’. In addition to the development step (‘In Progress’), this particular task board has two

verification steps: ‘Review’, meaning code review, and ‘Testing’. It is important to put verification steps into the development process, as user stories can be considered done only when their quality has been verified and validated. Some Agile teams do not follow the proper definition of ‘Done’ and omit testing steps on the task board, allowing tasks to be marked as ‘Done’ after the code has been produced without any verification steps.

The task board needs to be reviewed regularly to ensure the team is on track and every task will be finished on time. Agile teams regularly review the board during daily stand-up meetings, answering the three questions mentioned above with regard to their assigned tasks on the task board. If any issue has been identified as blocking a task, it is mentioned during the stand-up meeting and usually other team members volunteer to help after the meeting. For example, if a tester states that they cannot execute tests as there is no suitable test data in the test environment, and the tester does not have sufficient experience in generating test data, one of the other team members may offer to help in loading data and sharing the knowledge of how to do that in the future.

Another common example of blocking tasks may involve a defect in one of the product functions that is blocking the steps in test scenario. The tester should raise this issue during the stand-up meeting so that the team can look at reprioritising bug fixing activities to unblock test execution.

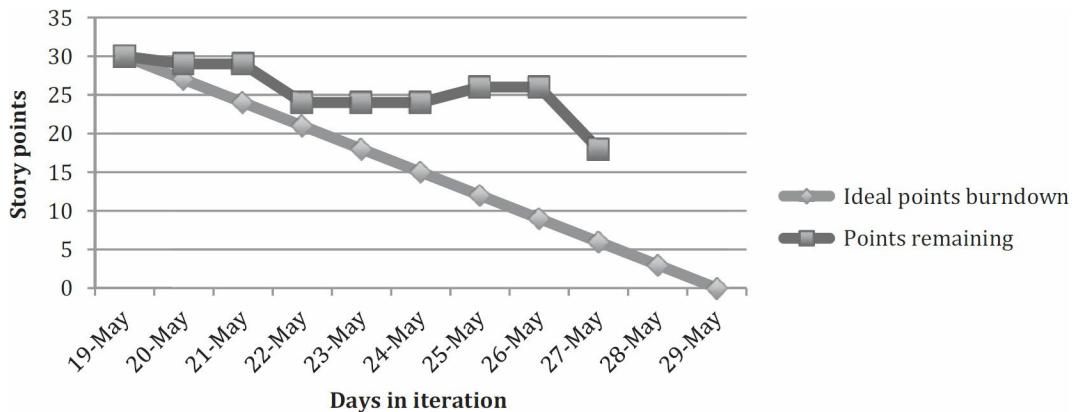
To check iteration and overall product progress, teams usually prepare burndown charts that show actual progress against the planned progress, as demonstrated in [Figure 2.8](#).

---

**Figure 2.7 Example of a task board**

Story	New	In Progress	Review	Testing	Done
Sprint Impediments					
(0 hours) 2027 WebApp - Login				2032 Login Web Form Jan Sabak Normal	2033 Permissions scheme Jan Sabak Normal
(0 hours) 2028 WebApp - New Transaction		2035 Form Layout - New Transaction John Smith Normal	2034 Form Layout - New Transaction Susan Brown Normal		2031 Form Layout Susan Brown Normal
(0 hours) 2029 WebApp - History of Transactions	2037 WebForm - History John Smith Normal	2036 History - Form Layout Susan Brown Normal			
(0 hours) 2030 Bugfixing Previous					

**Figure 2.8 Example of a burndown chart**



Burndown charts usually show two lines. One line ('Ideal points burndown') shows how many story points should be left to complete ('burn') at any single day of the iteration. The other line ('Points not accepted') shows a summary of the number of points belonging to tasks not yet done. If the actual line is above the ideal line, the team knows that they are not tracking to the estimates provided at the start of the iteration.

As we can see from the burndown chart example in [Figure 2.8](#), the team would have been alerted to a problem with their progress quite early as the

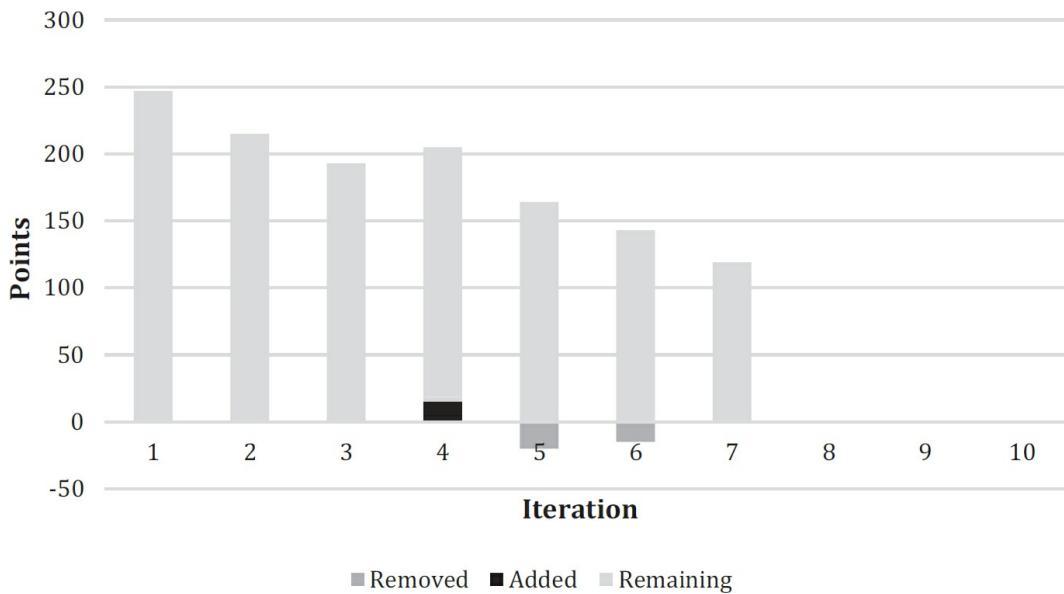
‘Points not accepted’ line is not trending to the ‘Ideal’ line. Burndown charts are a great way to track the team’s progress and identify and take necessary corrective action against any slippages; however, commentary may also be required, especially when the ‘Points not accepted’ line starts to increase instead of decrease. This could be due to a number of reasons, such as large volumes of defects when verifying user stories, or the team may have accepted additional user stories into the iteration. The burndown chart in this example shows that at the end of the iteration the team have only delivered half of the story points they signed up for at the start of the iteration. This chart clearly indicates that the team will need to take corrective action to fix their iteration velocity, to ensure the next iteration is sized more accurately.

Burndown charts can also show overall progress for the whole project (called the product burndown chart) or nearest release. This chart contains the size of the product features that remain to be developed (see [Figure 2.9](#)). It also shows changes in project scope, depicting added and removed features.

There are several types and styles of burndown charts; they differ in the way progress is visualised (see [Figures 2.8](#) and [2.9](#)) and in units in which project effort is measured. Units used in charts should be correlated to the units used in iteration planning. The most commonly used units are story points, number of tasks and ideal work hours, but some teams use artificial units, such as T-shirt sizes and even guinea pigs (as used in the company of one of the authors).

---

**Figure 2.9 Example of a product burndown chart**



To measure the overall product quality, Agile teams conduct satisfaction surveys to get an understanding of the users' experiences with the product and to assist in improving the overall quality of the product. These surveys aim to show how successful the team is in satisfying the stakeholders' business needs. They can ask following questions:

- Did the sprint result in added value to the product?
- Was the new functionality as expected?
- Do you think you will be using the new functionality?
- What is your overall satisfaction level with the sprint?

The other kind of survey teams may use is a team satisfaction survey, which measures the satisfaction of team members. These surveys show how the team is performing internally, whether people are working well together. The following questions can be asked in these surveys:

- Do you feel that the whole team shares responsibility for the success or failure of the sprint?
- Do you feel afraid to ask for help?
- Do you enjoy working together with other members of the team?

- What is the most annoying behaviour of the team?

On a daily basis, the team can track the mood of team members with a Niko-niko calendar ([Figure 2.10](#)). At the end of each day, team members record an evaluation of their mood during that day, which is generally represented by a facial expression on a colour-coded sticker. The first documented use of the Niko-niko calendar, or Niko-cale, was described by Akinori Sakata using a yellow smiley face for a good day; a red neutral face for an ordinary day; and a blue sad face for a bad day. Other implementations of Niko-niko calendars have used the traffic light colour coding to represent the team's mood, using blue or green sticker dots to represent a good day; yellow sticker dots to represent an ordinary day; and a red sticker dot to represent a bad day. When implementing a Niko-niko calendar, either method above can be used, it all depends on which method suits the team. A Niko-niko calendar helps to reveal the change in mood of the team and individual team members, and can assist in preventing major interpersonal issues.

**Figure 2.10 Example of a Niko-niko calendar**

August	15	16	17	18	19	20	21	22
John S.	😊	😊	😊	😊	😊	✗	✗	😊
Susan B.	😊	😐	😊	🙁	😐	✗	✗	😊
Paul K.	😊	😊	😊	😐	🙁	✗	✗	🙁

**Note:**

Niko-niko

calendar:

<https://agilealliance.org/glossary/nikoniko/> and [www.geocities.jp/niknikocale/](http://www.geocities.jp/niknikocale/)

Agile teams may also use other metrics of work process similar to those used in traditional development methodologies, such as:

- test pass/fail rates;
- defect discovery rates;
- confirmation and regression test results;
- defect density;
- defects found and fixed;
- requirements coverage;
- risk coverage;
- code coverage;
- code churn.

These metrics can be used by teams to enable them to assess the product's quality and work progress throughout the iteration. They give the team feedback on their work. Tools should be used to gather metrics automatically to enable the team to focus on doing their work, rather than reporting it.

Apart from showing product quality and work status, metrics can be used to assess process and identify process improvement opportunities. They can be analysed by the whole team during retrospectives to decide on changes in development and testing processes. Gathered metrics should be relevant, useful and understandable by all.

Metrics should not be used to reward, punish or isolate any team members, as the whole team is responsible for the quality of the product. Isolating or scapegoating team members or roles is against Agile principles.

## **2.2.2 Managing regression risk with evolving manual and automated test cases**

Agile projects are completed in multiple short iterations, with each iteration adding functionality to the product. Changes to previously developed features can also occur in later iterations based on customer feedback or if the project scope changes.

Due to the constant change experienced in an Agile environment, the scope of testing also increases with each iteration, requiring testers to develop new test cases and change or retire existing test cases from previous iterations to keep pace with the developed functionality. In addition to keeping up with the changes, testers also need to check whether regression has been introduced that may destabilise the quality of the product.

As the product evolves with each iteration, the number of tests to be executed significantly increases. An iteration's time span is constant, so from iteration to iteration the overall number of tests constantly grows and testers have correspondingly less time per test case in each iteration. This necessitates the automation of as many test cases as possible and as practical. Automated tests give fast feedback to developers and testers on the quality of the product. Automated tests can be run during the night, providing the team with test results at the beginning of the next day. They can also be run continuously throughout the day with help of CI tools, which can also perform other tasks, such as static analysis or automated deployment to test environments.

To ensure easy access and maintainability of tests by all members of the team, they should be stored in a configuration management tool. This ensures that test scripts and the test framework are correctly version controlled, allowing review of history of changes and, in some cases, correlating the automated test cases with the product code (refer to [Section 2.1](#) for more information on configuration management).

During an iteration, testers must design and execute tests for new features, but at the same time they also need to check how the new features interact with or impact on other features that were implemented in previous iterations.

Testers should also check if the changes in code have revealed defects that may have been previously hidden. Based on this, it is imperative that the team performs a significant amount of regression testing. The timeframe within an iteration seldom allows manual repetition of all test cases, so testers should identify and select the necessary tests for a suitable regression test suite, automating as many of them as possible. Test cases from previous iterations should be regularly reviewed and checked to ensure they are still relevant, and then updated or retired when required. If regression test cases are not reviewed, updated or retired regularly, it is possible that they will produce false positives or false negatives in later iterations where the functionality they relate to has changed, and this can be quite time-consuming to analyse.

While choosing tests for a regression test suite, a tester should first consider risk coverage. The regression test suite should consist of tests that touch every risk item according to risk analysis. Only when all risks are covered should testers add more test cases according to risk level. An additional set of test cases should be identified to be included in the build verification test (BVT), which (per the ISTQB® Glossary) is a set of automated tests that validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs and it is run on every new build before the build is released for further testing.

As previously mentioned, the nature of Agile development necessitates the need for test automation. Agile projects rely on tight feedback loops, so the more test cases that can be automated, the better. However, a balance is required when identifying the test cases to automate. It can be quite an expensive exercise to automate test cases for features that constantly experience significant changes – it can be more productive and cost-effective to hold off automating these test cases until that functional area or feature has

been stabilised.

According to the test pyramid (refer to [Section 3.1](#)), tests should be automated at all levels. Programmers should automate test cases at a unit test level. This will give them feedback in seconds. Automated system and integration tests give feedback in hours or daily, when these tests are executed per the build and CI schedule implemented within the organisation.

Choosing whether to automate a test or not is just choosing the way it will be executed. Any test case, manual or automated, should be well designed. Poorly designed test cases (for example, tests without expected results) when automated may lead to incorrect or poor test results, which may give a false sense of security in the level of product quality.

Teams need to remember that automated test cases are also software, therefore good coding and design practices should be observed. As with all software development, if the design of the test automation is poor or not well constructed, then they run the risk of additional effort being required to update and maintain the test automation. In some cases, the effort to maintain poor test automation can be greater than writing new automated test cases from scratch.

### **Author example**

I have been involved in projects where they began with capture-replay test automation, which started with one script, and then another, and another. After a while, there were more than 100 scripts, which were hard to maintain. As a team, we decided that if there were more than 20 automated test cases in the project, then using best practices such as keyword-driven test automation and page object patterns<sup>1</sup> were a must. This assisted the team with building robust and maintainable test automation, and allowed us to quickly adapt test cases with each code change throughout the product's evolution.

Test frameworks should also integrate with CI tools. They should possess the means to choose which tests are to be executed. In this way, appropriate test

suites (for example, BVT, smoke tests or full regression tests) can be run automatically and provide feedback on product quality at chosen time intervals. Test results show which features have defects and which are in good condition, so CI helps to identify regression risks.

One of the principles of Extreme Programming is ‘collective code ownership’. This means that every programmer in the team can refactor the code that was written by other programmers (‘refactor mercilessly’ is another principle to consider). This also means that the project must have a comprehensive suite of automated tests (especially unit tests). The CI tools then run these test cases against all code changes, and the programmer changing code is obliged to bring the code to the state where it passes all tests. This corresponds well to the testing principle of early testing and the Agile principle of ‘test early, test often’ and helps to manage the risk of regression.

CI tools also perform static analysis of code, automated unit testing, automated system and integration testing, and automated deployment to test environments. Testers can then execute their manual tests, including exploratory tests, on stable software without having to deal with low-level technical problems such as unhandled exceptions, integration failures, inadequate decision coverage, memory leaks and so forth that should have been found in earlier stages.

When any test fails in an Agile project, the results should be investigated immediately to identify and fix the underlying cause of the failure, so as not to allow technical debt to accumulate. High technical debt means higher costs of maintenance and problems with changing code, as future code is being built upon a possibly unstable code base. In some instances, the investigation into the test failure can identify that it was not actually a bug, but the test was out of date, possibly due to changes in requirements or the test environment, so the testers would need to fix it or retire it and design another test that is

more suitable. In some cases, the analysis of incidents can identify that development processes may need to be improved to avoid similar mistakes or defects in the future, which is all part of the feedback loop encouraged in Agile development teams.

In some cases, the manual or automated execution of regression tests can take too long to run within the tight timeframes of an iteration. Iterations in Agile projects are usually two to four weeks in length, i.e. 10 to 20 working days, which can be challenging when trying to run system and acceptance level regression tests for large or complex systems. For regression test cases that can take several days or weeks to run, some Agile teams remove the lengthy regression tests from feature development iterations and create a dedicated iteration, sometimes known as a hardening iteration, for the sole purpose of performing a full regression test on the software, including fixes to defects found from the regression tests, prior to release. The same may happen for the development of automated tests. Test automation may take too much time and effort to be completed during an iteration. In some Agile teams, iterations are created for the purposes of developing automated tests. In other Agile teams, a test automation specialist or team continuously develops test automation in parallel with all other tasks within the iteration.

Regardless of the way regression tests are performed, in Agile projects they play an important role by enabling the team to assess and manage risk in a very fast-paced development environment. Keeping the regression test suites up to date allows the project team to mitigate many product quality risks (for example, defects in the rendering of web forms due to changes in common style sheets, or functional defects in a web application due to changes in underlying services). It is critical that updates to regression tests keep pace with changes to the product. It is also important to have as many automated test cases as possible, which enables faster and higher execution frequency. To ensure the team gains the benefits of test automation, a balance needs to

be struck. The automated test suite should be aligned with risk analysis and impact analysis so as not to let it grow uncontrollably, which can make maintenance more costly, and at the same time ensuring that it provides sufficient coverage of existing functionality. Some Agile development approaches (for example, Extreme Programming (XP)) require the creation of automated tests for each found defect to avoid their future reoccurrence. By adopting this approach, it enables the team to grow the automated regression test suite at the same time as ensuring coverage of error-prone areas of the system.

In addition to test automation, the following testing tasks may also be automated:

- test data generation;
- loading test data into systems;
- deployment of builds into the test environments;
- restoration of a test environment (for example, the database or website data files) to a baseline;
- comparison of data outputs.

Some of these tasks can be included as part of the test automation framework (for example, test data generation, comparison of data outputs) or as individual tasks, depending on the ability of the test tool. Some of these tasks (for example, deployment of builds) require the use of specialised tools such as CI tools or configuration management tools.

## **Summary**

In this section, we have looked at the fast-paced nature of Agile development, and identified the importance of communicating the status of testing to all project stakeholders to enable the team to make informed decisions on the direction of product development and project outcomes. We have looked at

various ways of being able to communicate test progress and product quality metrics, with all methods identified having the common goal of ensuring real-time status visibility to all project team members and stakeholders.

We further delved into the challenges of constant product changes in each iteration and the necessity for test cases to evolve with each new iteration as new features are built upon existing features from previous iterations. We also identified the regression risks associated with Agile development and the importance of automating regression testing to reduce these risks, and ensure that in each iteration testers are able to balance the need for constant increased regression testing with the need to verify and validate new feature development.

## Test your knowledge

The answers to these questions can be found in the Appendix.

### Agile Tester Sample Questions – [Chapter 2 – Section 2.2](#)

#### Question 1 (FA-2.2.1) K2

Which of the following would NOT be an expected response from a team member during the daily stand-up meeting?

#### Answer set:

- A. My tasks to verify the login screen and Create New User are currently blocked by an integration issue with our authentication server.
- B. Since the last meeting, I have successfully verified the Add to Cart, Modify Cart Items and Remove Cart Items features, and am part of the way through verifying the Cancel Order feature.
- C. After we investigated the issues with the authentication server, we

found there was an issue with the website service account being on a different domain from the authentication server. The network team is unsure how to fix this as it needs to be cross-domain accessible. Does anyone in the meeting have any suggestions?

- D. If the issue with the authentication server is resolved today, I intend to finalise verification of the Login screen feature, otherwise I will move onto verifying the Postage and Shipping tasks.

### **Question 2 (FA-2.2.2) K2**

During an iteration planning session, you are allocated a task to review test cases from previous iterations. Which of the following would provide the BEST reason for this task?

#### **Answer set:**

- A. All test cases from previous iterations need to be automated
- B. The product is constantly changing and previous test cases may no longer be relevant
- C. To ensure the test cases have sufficient information to be able to be tested by others
- D. To fill time while the developers code new features, so not sitting idly waiting for new features to test

## **2.3 ROLE AND SKILLS OF A TESTER IN AN AGILE TEAM**

The learning objectives for this section are:

FA-2 (K2) Understand the skills (people, domain and testing) of a tester in an Agile team

FA-2 (K2) Understand the role of a tester within an Agile team

Are testers required on an Agile team? We established in [Section 2.1](#) that there is a need for independent testing on Agile teams, now we are going to explore what skills a tester needs to be successful on an Agile team, and also delve further into understanding the role of a tester within an Agile team. We will look at how we can transfer our skills as traditional testers into an Agile environment, and what skills we need to build upon or gain to be able to successfully collaborate and contribute to the Agile team.

### **2.3.1 Agile tester skills**

In an Agile team, testers must closely collaborate with all other team members and with business stakeholders. This means that a tester has more activities than in a traditional development team or project.

This section presents the skills and behaviours expected of a tester in an Agile team, independent of the Agile approach. This list of skills and behaviours is specified on the ISTQB® Foundation Level Extension Agile Tester syllabus ([www.istqb.org/downloads/syllabi/agile-tester-extension-syllabus.xhtml](http://www.istqb.org/downloads/syllabi/agile-tester-extension-syllabus.xhtml)). The skills include:

- Being positive and solution-oriented with team members and stakeholders.
- Displaying critical, quality-oriented, sceptical thinking about the product.
- Actively acquiring information from stakeholders (rather than relying entirely on written specifications).
- Accurately evaluating and reporting test results, test progress and product quality.
- Working effectively to define testable user stories, especially acceptance criteria, with customer representatives and stakeholders.
- Collaborating within the team, and working in pairs with

programmers and other team members.

- Responding to change quickly, including changing, adding or improving test cases.
- Planning and organising their own work.

## **Testing as an activity on an Agile team**

Previously, we have explained that before implementing a user story, it must reach the state of being ‘ready’. It is good practice to have a clear understanding of what needs to be performed for the user story to be considered ‘done’, and have the required tests ready before implementing the user story. In this case, we can consider that the team has correctly understood the goals of what has to be developed.

As discussed in [Section 2.1.5](#), people with specific skillsets should perform specific roles in the team, such as programmers designing and coding, and testers testing. In this case, from an efficiency perspective, if a person is trained in testing techniques and has the mindset of a tester, then it would be expected that they take on the testing role, and assign themselves to testing tasks, as they are the appropriately skilled person to perform these activities. They will have the technical excellence and ability to implement good test design and enhance the team’s agility, which meets Agile principle 9, ‘Continuous attention to technical excellence and good design enhances agility’, one of the 12 principles behind the Agile Manifesto. For example, organising and/or implementing exploratory testing is a part of the tester’s activities. An efficient tester will also use a quality characteristic classification, such as ISO 9126, to help to better identify and sort the business needs into functional and non-functional categories. A person with testing skills, as explained in the ISTQB® Foundation syllabus, will help the team to reach the level of software quality it expects in promoting quality assurance activities in parallel to designing, executing and reporting test cases

(for example, test coverage instead of test execution reporting). This will ensure transparency and professionalism for the stakeholders.

## **Business and functional testing skills**

One of the Agile values is ‘Working software over comprehensive documentation’. Business and functional testing, and risk prevention, are fundamental testing activities prior to releasing software components and before checking essential non-functional quality characteristics such as performance or usability.

As explained in [Section 1.2](#), the creation of user stories is a collaborative approach, where the tester works alongside the team creating test cases and contributing to the acceptance criteria by asking the business representative open-ended questions. To be able to be successful in an Agile team, the tester must have a level of business analysis skill to assist in understanding what the user representative expects. The tester role on an Agile team needs to actively acquire information from stakeholders rather than relying entirely on written specifications. However, in saying that, in some organisations that develop embedded software or safety-critical engineering solutions, there are still formal written specifications. These documents are needed to formalise the information to comply with regulatory requirements.

Once the acceptance criteria have been defined in either a user story or formal requirements document, the tester will commence creation and execution of their test cases, also focusing on the non-functional quality characteristics. Testers continue to use the same test techniques as they would on a traditional development team, such as black-box test techniques, to demonstrate the acceptance criteria and correct implementation of the business conditions of the user story.

Testing tasks can be done by either a dedicated tester role, or various people with differing skillsets. For example, a test automation engineer/specialist

may work on automating the test cases, a person with business analysis experience could work on defining the acceptance test cases, and developers could assist with test automation frameworks and defining the technical tests.

## **Technical testing skills**

The environment of an Agile team is technical as the team continually builds upon already developed software components. A tester needs to be more efficient in their testing efforts by implementing automated testing to cover regression testing risks (refer to [Section 2.2](#)). To be technical does not mean that the tester is inevitably a developer, it means that the tester must understand what is going on in the team and also understand development concepts. The tester is a member of the Agile team, participates in the daily stand-up meeting and will present any testing impediments and have to understand the impediments being experienced by other team members in order to better test the solution that will emerge.

To be technical also means that a tester has to be able to propose:

- efficient ways of testing, including efficiencies in test automation;
- different ways of describing test cases, including TDD and BDD;
- different approaches to testing and development tasks, such as being able to pair with developers, being able to create mocks (a code object created to simulate the behaviour of a real object in a controlled way) or being able to virtualise a service.

Because of the incremental and iterative nature of an Agile development approach, regression testing can be a large overhead, but it is also a critical task to manage the regression risk (refer to [Section 2.2](#)). Given the large amount of code churn in an Agile project, to ensure functional and technical risks are managed adequately, regression tests need to be automated. For this reason, testers in an Agile environment need to have test automation skills.

Sometimes this test automation activity is integrated in acceptance test-driven development (ATDD) (refer to [Chapter 3](#) for more information on ATDD).

Tools are required to automate tests, which are based on various programming languages. Even keyword-driven test frameworks need to be coded. At a minimum, testers should have scripting skills to be able to script test cases that are included in an automated test framework. Testers need to be able to switch from one tool or language to another, depending on the technology used on the project. Testers also need to understand the way CI or deployment works, how a software component is built, and how software configuration management works and so on, in order to be a contributor to these tasks (as described in [Section 2.1](#)).

When using a test-first approach in an Agile project, especially when using TDD or BDD, the tester is a lot closer to the code and is also working closely with the developer; for example, in pair programming. Even if the tester proposes their test cases under the ‘Given-When-Then’ syntax, they need to communicate with the developer in technical terms in order to understand what the developer can automate (refer to [Chapter 3](#) for more information on TDD and BDD).

In some Agile teams, the tester is identified as the quality assurance (QA) person. This role relates to more than just testing activities. All members of an Agile team are responsible for quality; however, each member of the team sees quality differently. A tester in an Agile team must have, as a minimum, the skills described in Section 5 of the ISTQB® CTFL syllabus. These are:

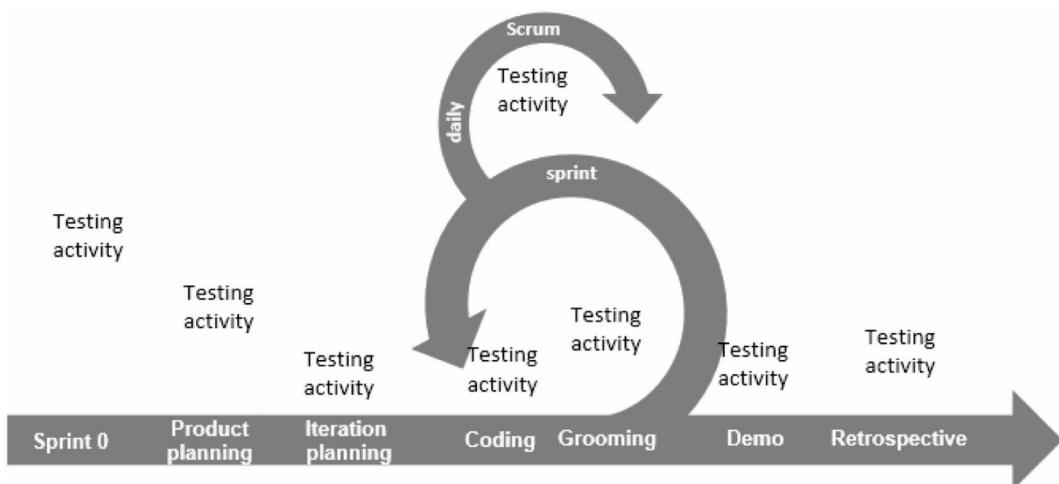
- Reviewing and contributing to test plans.
- Analysing, reviewing and assessing user requirements, specifications and models for testability.
- Creating test specifications.
- Setting up the test environment.

- Preparing and acquiring test data.
- Implementing tests on all test levels, executing and logging the tests, evaluating the results and documenting the deviations from expected results.
- Using test administration or management tools and test monitoring tools as required.
- Automating tests (may be supported by a developer or a test automation expert).
- Measuring performance of components and systems.
- Reviewing tests developed by others.

As mentioned previously, these are the minimum tasks and skills expected of a tester, and it is recommended that testers incorporate the tasks and skills described in this book to enhance their ability to provide greater value to the Agile team.

As testing is required in all aspects of an Agile project, a tester is sometimes required to act as a quality coach to other team members within these activities, as demonstrated in [Figure 2.11](#).

**Figure 2.11 Testing activities throughout iteration activities**



## **Interpersonal skills**

As a team member, the behaviour of a tester is important because working software is the primary measure of progress. The tester must propose proactive solutions towards efficiency. Facing impediments, a tester must stay positive and propose solutions instead of criticising. They also need to promote software quality within the team and provide software testing and quality coaching.

### **Author example: wrong behaviour**

I remember a test manager in an Agile team who had to organise the tests for all iterations. He had to plan tests through iterations and design end-to-end tests when features were assembled. The difficulty for him was to adapt the plan when the backlog was changed. He had been implementing very detailed algorithms and lost time having to redraw them when backlogs did change. He also did not feel the atmosphere and morale within the team was very good, and in fact felt he was not really working within the team. He asked people for documentation to review when the team had technical issues to solve. He complained that the project did not progress, that people did not provide documentation and that the product owners were always changing the product backlog, order and content. He did not participate in iteration planning or release planning sessions.

He did not understand that he had to adapt his documentation and behaviour to align with the Agile team to enable him to be able to respond to the changes, and was missing the value ‘Responding to change over following a plan’. This is a risk when people switch from a traditional development lifecycle to an Agile team.

Since his behaviour and actions were not aligned with the Agile team, he and the team suffered communication issues, which endangered the success of the project.

In some open spaces the Agile values and principles are posted clearly on the walls to ensure all team members are aware of them and can evaluate their own behaviour against them.

### **Author example: good behaviour**

In another organisation, there were two testers integrated in an Agile team. These two people felt really engaged with the team from a quality perspective and gave their support to the other team members. They were involved in all of the Agile meetings. In planning sessions and user story

grooming (sessions to review and make improvements to the product backlog), they brought along their knowledge in testing activities to define acceptance criteria and help with estimating the effort on user stories. As there was no coach, they assumed the role of coach. They helped developers to implement BDD. In addition, they continued in their role as testers, creating and executing test cases and reporting test coverage. They progressively improved their approach over many iterations to achieve a high performing self-managed team – during the first iterations, the testing point of view was not taken into account, there were regression issues and a lot of user stories not ‘done’ at the end of the iteration.

Considering the Agile mindset, which enables the team to self-organise and bring the most added value they can, they worked with the team and proposed solutions to improve product quality that were accepted by the team. So, progressively, retrospective after retrospective, they helped the team to improve the quality of the product with each iteration.

### **2.3.2 The role of a tester in an Agile team**

Within an Agile team, each team member is responsible for product quality and plays a role in performing test-related tasks. Each one in the team may consider quality differently in terms of their focus and mindset (as described in Section 1.5 of the CTFL syllabus). For example, a developer will prepare and execute unit tests and improve their code with static analysis tools, which is one of many mechanisms of contributing to quality; a business-oriented person will focus on business tests, which would be their contribution to quality; and an architect will possibly define the need for performance tests, which would be their contribution to quality. All of these are valid contributions to quality; however, an independent tester on the Agile team would be able to provide an overarching viewpoint to all of these contributions to formulate the strategy for the overall product quality.

Testing, as well as product and process quality activities, occur during the whole project, release and iterations. Some of the testing activities are to prevent defects and others to test what has been developed. We now discuss the activities in which the tester generates and provides feedback to the Agile team.

## **During the test strategy**

Before its implementation, the tester needs to understand the test strategy. As change is expected in an Agile project, the test strategy will need to be continuously updated. The tester may be a contributor during the first stages of a project, and can participate in the design of this test strategy. Here, a tester can bring their knowledge about test techniques and definitions of the different statuses, especially around the ‘definition of ready’ (DoR) and ‘definition of done’ (DoD). As an example of how to get to the status of DoR, they may propose that tests must exist before implementing a user story, otherwise the risk is to not have finished development (not ‘done’) or an increase in defects may be found during the later iterations.

The test strategy must also define **what kind of tests** and **quality characteristics** need to be tested, and when. For example, a tester will explain why they need some modules to execute performance testing or end-to-end testing of an epic developed in several iterations. As in all test strategies, the tester in an Agile team helps to estimate the **overall test effort**, the **technical resources** they need, if **specific skills** are needed and the **roles of each internal and external team member**.

With regard to technical resources, the tester describes and discusses the test environment and the tools they need. They may discuss the opportunity of using a specific tool, such as a ticketing tool, for user story/requirement and defect management and the effect on the tests. They can help to select tools, such as traditional test management tools and Agile tools for BDD or ATDD. They also propose associated test-first approaches. This is important, because they will generally take on the responsibility of configuration management of test cases.

Also during the test strategy phase, they need to identify the training needs of the team in test techniques. For example, some new tools with specific scripting techniques can be implemented for BDD.

They will need to ensure also that the appropriate testing tasks are scheduled during release and iteration planning sessions.

## **Before implementing**

One of the strengths of an Agile approach is the backlog definition. If the product backlog is sufficiently defined, the team will have a better understanding of what has to be implemented and then will avoid future defects. The tester helps in that by clarifying requirements with examples during requirement workshops, grooming or release and iteration planning sessions, especially in terms of testability, consistency and completeness with developers, business stakeholders and product owners. The role of the tester is one of the three of the ‘power of three’ or one of the ‘three amigos’. For more information on these, refer to [Section 1.1.2](#).

During iteration planning, the tester participates in the evaluation of the effort needed to implement the user story or requirement, which must include the testing tasks, including creating test data, which can be time-consuming.

## **During the implementation of a requirement**

During implementation, an example of the testers’ activities for the iteration can include:

- Configuring the test environments and creating test data sets.
- Designing tests depending on the criticality of the requirement.
- Executing tests and helping other team members to create and execute tests, possibly also assisting developers with unit tests.
- Performing static analysis.
- Selecting tests for regression testing purposes.
- Executing regression tests (both automated and manual) or ensuring they are performed.

- Automating test scripts.
- Reporting defects and working with the team to resolve them.
- Supporting the team in the use of testing tools.
- Attending the daily meeting and reporting on progress and any impediments.
- Moving the user story or requirement task on the board to ‘done’ once all tests have passed.
- Measuring and reporting test coverage across all applicable coverage dimensions.

In all these activities, the tester needs to be flexible and receptive to changes: changing, adding or improving their test cases.

### **During the retrospective**

The Kaizen spirit of continuous improvement, as used by Lean teams, leads the retrospective and enables Agile teams to constantly look at ways to improve their processes. Once the main processes and activities are set up, the team needs to tune them and focus their attention on quality and tests. Like craftsmen, the team improves with new ideas, and testers should proactively suggest improvements. For example, with regard to test processes, decisions can be taken to stop using an inefficient testing tool, or to use a scripting tool to create test data, or to better take into account the test effort during the scoring. Testers should be encouraged to provide feedback on all aspects of the Agile team when they see the possibility of improvements within the team.

### **Test-related organisational risks**

Some of the test-related organisational risks that Agile teams may encounter include:

- Test activities must be well defined in the test strategy in order to keep the advantage of test independence. In some organisations, in which people work closely together, a risk is not to have consensus within the team and the same vision about how the software works, as this can lead to the possibility of not detecting defects. If the tester pairs with a developer, they can run the risk of having the developer influence the tester on what has to be tested, based on what has been coded rather than what is defined in the user story or requirement. In this scenario, the tester may lose the appropriate tester mindset and independent perspective. A tester must stay independent and maintain their critical, quality-oriented and sceptical approach when testing the product.
- Testers become tolerant of, or silent about, inefficient, ineffective or low-quality practices within the team. This can occur if the team becomes too self-sufficient or if a coach does not change the techniques used during retrospectives with new ones, such as innovation games Speed Boat<sup>2</sup> or Plus/Delta.<sup>2</sup>
- Changes must be contained within the release backlog, and not the iteration backlog, as testers will not be able to keep pace with the incoming changes in time-constrained iterations. Within an iteration, in Scrum for example, changes in the content of the iteration should not occur. If they do occur, testers will be overburdened by time pressure as the workload had previously been estimated to fit in the time box. Different solutions, such as delegating regression tests to other team members, distributing the test automation to parallel teams and so on, can mitigate this risk.

## **Summary**

Following on from [Sections 2.1](#) and [2.2](#), in this section we have delved into understanding the skills and role of a tester in an Agile team. We have also

identified that the testing skills established in traditional testing build the foundation of the skills required of a tester in an Agile team. We also looked at the behavioural aspect of being an Agile tester: the need to be able to respond quickly to change; the need to work more collaboratively with other members of the Agile team; the benefit of having a positive and solution-oriented approach; the ability to display initiative and being proactive in acquiring information that may be required; and the need to be self-sufficient and self-organised.

In this section we have also reinforced the need for the tester role on an Agile team, and the benefits that an independent tester can provide to the team within the various testing activities throughout the entire Agile lifecycle – involvement in every aspect of an iteration, every aspect of a release.

## **Test your knowledge**

The answers to these questions can be found in the Appendix.

### **Agile Tester Sample Questions – [Chapter 2 – Section 2.3](#)**

#### **Question 1 (FA-2.3.1) K2**

Which of the following statements is TRUE within an Agile team?

#### **Answer set:**

- A. As the main goal is working software, testers must also have development skills to be cross-functional
- B. Testers should be results oriented and should actively participate in all Agile team activities
- C. Testers must be technically oriented so they are able to automate tests
- D. Testers should only be business oriented as the developers are

technical

### Question 2 (FA-2.3.2) K2

Which of the following statements is NOT TRUE within an Agile team about the tester's role?

**Answer set:**

- A. Testers participate in the definition of the test strategy
- B. Testers create written test cases when exploratory tests are not sufficient to cover the risk
- C. Testers maintain the iteration backlog as it is a low-level activity
- D. Testers are in charge of organising and conducting exploratory tests when needed

## REFERENCES

Cunningham, W. (2009) Done is Dead. Available at <http://forage.ward.fed.wiki.org/view/done-is-dead> (accessed April 2017).

## FURTHER READING

Beck, K. (2000) *Extreme Programming Explained: Embrace change*, 1st edn. Addison-Wesley Professional, Boston, MA, USA.

Crispin, L. and Gregory, J. (2008) *Agile Testing: A practical guide for testers and Agile teams*. Addison-Wesley Signature, Boston, MA, USA.

Poppendieck, M. and Poppendieck, T. (2003) *Lean Software Development: An Agile toolkit*. Addison-Wesley Professional, Upper Saddle River, NJ, USA.

## WEBSITES

[www.mountaingoatsoftware.com](http://www.mountaingoatsoftware.com)

[www.innovationgames.com](http://www.innovationgames.com)

<https://www.agilealliance.org>

<https://www.kaizen.com/about-us/definition-of-kaizen.xhtml>

## NOTES

- 1 A page object pattern represents or models the screens of a web application as a series of objects and encapsulates the features represented by a page within the test code. This means that if the user interface (UI) changes, the fix to the test cases only needs to be applied in one place (<https://github.com/SeleniumHQ/selenium/wiki/PageObjects>; accessed May 2017).
- 2 The goal of the Speed Boat innovation game ([www.innovationgames.com/speed-boat/](http://www.innovationgames.com/speed-boat/); accessed May 2017) is to identify what customers do not like about your product or service with a view of finding fresh new ideas for product changes to address the customer's most important concerns.
- 3 The goal of the Plus/Delta innovation game ([www.innovationgames.com/plusdelta/](http://www.innovationgames.com/plusdelta/); accessed May 2017) is to collect constructive criticism to improve future events by taking negative feedback and transforming it into useful information.

### **3 AGILE TESTING METHODS, TECHNIQUES AND TOOLS**

By Rex Black, Istvan Forgács, Kari Kakkonen and Jan Sabak

In [Chapter 3](#), we get into the day-to-day operational details of what an Agile tester must do. The first section addresses Agile testing methods, including test-driven development, acceptance test-driven development and behaviour-driven development. We put those methods in the context of two ways to think about both manual and automated testing, the concepts of the test pyramid and the testing quadrants, and relate those two concepts to the traditional ISTQB® concepts of testing levels and testing types. To make these testing methods more specific and concrete, we use the role of the tester in a Scrum team.

[Section 3.2](#) addresses how to determine, allocate and prioritise testing for Agile projects. It starts by showing you how to adapt common testing best practice and risk-based testing to iterative Agile lifecycles. It then shows you that the insights from risk-based testing can be used to estimate test effort within each iteration.

In [Section 3.3](#), we move from the test-related planning and analysis activities

associated with risk analysis and estimation to the design, implementation and execution of tests. We cover how to identify and interpret the information you need to create and run tests, including working with business stakeholders so that the acceptance criteria allow you to create tests with clear pass/fail results. We address the use of black-box test design techniques to transform user stories and their acceptance criteria into test cases. We then return to the concept of acceptance test-driven tests, showing how to translate the black-box test designs into acceptance test-driven test cases that verify user stories and their acceptance criteria. Finally in this section, we cover the use of exploratory testing to validate behaviour against user expectations and needs.

In the final section of this part, we discuss how tools fit into Agile methods. This discussion complements the ideas presented in the ISTQB® Foundation Level syllabus. Many tools used in traditional lifecycles apply in Agile lifecycles, but in some cases they are used differently, which we discuss. In addition, some tools are uniquely associated with Agile methods, and we discuss those tools as well.

### **3.1 AGILE TESTING METHODS**

The learning objectives for this section are:

FA-3.1.1 (K1) Recall the concepts of test-driven development, acceptance test-driven development

and behaviour-driven development

FA-3.1.2 (K1) Recall the concepts of the test pyramid

FA-3.1.3 (K2) Summarise the testing quadrants and their relationships with testing levels and testing types

FA-3.1.4 (K3) For a given Agile project, practise the role of a tester in a Scrum team

Development and testing in an Agile lifecycle model does not happen with

good will only. The Agile team need access to methods and tools that will help in the testing work. Moreover, the chosen methods must be suitable to the quick feedback cycle of the Agile project. They must enable the testing not only to keep up with the speed of the Agile project, but also to help the Agile project meet its goals and enable it to be more Agile.

All Agile team members, including developers and business stakeholders, can and should participate in testing, but the Agile tester brings knowledge, experience and focus to their testing tasks. They can guide others in the testing work with their profound knowledge of testing, acting as quality coaches for the rest of the team.

Many testing techniques are suitable for any lifecycle model; however, some are more suited to Agile projects. Indeed, some were born in Agile projects and are now also used in more traditional projects. In Agile projects, you can also find and use many techniques that are introduced in [Chapter 4](#) of the ISTQB® Foundation Level syllabus.<sup>1</sup> In this section, we describe some testing techniques and ways of thinking that originated in Agile development.

### **3.1.1 Test-driven development, acceptance test-driven development and behaviour-driven development**

The traditional way of developing code is to write the code first and then test it. Some of the major challenges of this approach are that testing is generally conducted late in the process and it is difficult to achieve adequate test coverage. Test-first practices can help to solve these challenges. In this environment, tests are designed first, in a collaboration between business stakeholders, testers and developers. Their knowledge of what will be tested helps developers to write code that fulfils the tests. A test-first approach allows the team to focus on and clarify the expressed needs through a discussion of how to test the resulting code. Developers can use these tests to guide their development. Developers, testers and business stakeholders can

use these tests to verify the code once it is developed.<sup>2</sup>

A number of test-first practices have been created for Agile projects, as mentioned in [Section 2.1](#). They tend to be called X-driven development, where X stands for the driving force for the development. In test-driven development (TDD), the driving force is testing. In acceptance test-driven development (ATDD), it is the acceptance tests that will verify the implemented user story. In behaviour-driven development (BDD), it is the behaviour of the software that the user will experience. Common to all of these approaches is that the tests are written before the code is developed, i.e. they are test-first approaches. The approaches are usually better known by their acronyms. This section describes these test-first approaches and information on how to apply them is contained in [Section 3.3](#).

TDD was the first of these approaches to appear. It was introduced as one of the practices within Extreme Programming (XP) back in the 1990s.<sup>3</sup> It has been practiced for two decades and has been adopted by many software developers in Agile and traditional projects. However, it is also a good example of an Agile practice that is not used in all projects. One limitation with TDD is that if the developer misunderstands what the software is to do, the unit tests will also include the same misunderstandings, giving passing results even though the software is not working properly. There is some controversy over whether TDD delivers the benefits it promises. Some, such as Jim Coplien, even suggest that unit testing is mostly waste (Coplien, n.d.).

TDD is mostly for unit testing by developers. Agile teams soon came up with the question: What if we could have a way to get the benefits of test-first development for acceptance tests and higher-level testing in general? And thus ATDD was born. There are also other names for similar higher-level test-first methods; for example, specification by example (SBE) from Gojko Adzic.<sup>4</sup> Later, Dan North wanted to emphasise the behaviours from a business perspective, leading him to give his technique the name BDD.<sup>5</sup>

ATDD and BDD are in practice very similar concepts.

Let's look at these three test-first techniques, TDD, ATDD and BDD, more closely.

## **Test-driven development**

TDD is a method whereby unit tests are created, in small incremental steps, and, in the same small incremental steps, the code is created to meet those tests. Metaphorically, think of how a bush's shape can be made orderly and in conformance with desired behaviour by the use of a lattice-like frame (i.e. a trellis). These unit tests allow software developers to verify whether their code behaves according to their design, both as they develop the unit and after making any changes to the unit. This provides a level of confidence that leads many developers to stay with TDD once they have become accustomed to the process. However, other developers find the process too tedious or cumbersome, and instead create and run their unit tests after coding.

TDD involves first writing a test of the expected low-level functionality, and only then writing the code that will exercise that test. When the test passes, it is time to move to the next piece of low-level functionality. As you grow the number of tests and code base incrementally, you also need to refactor the code frequently. If you do not refactor, you may end up with 'spaghetti' code that is neither maintainable nor understandable. Some people would rather do lots of architectural design first, and you can do that as well, to some extent, to avoid too much refactoring. However, the strength of TDD is that you only develop the minimum code that is required to pass the existing unit tests, and only then move on to the next piece of test and code. This way you avoid unnecessary code. Your code will be lean, fast and maintainable. Debugging will be very easy as you have only a small code increment to look at (Lehtonen et al., 2014).

The tests need to be automated, usually in a test framework such as JUnit,

CppUnit or any of the other xUnit family of frameworks (where x stands for any programming language). We'll discuss these unit testing frameworks further in [Section 3.4](#). Without automation, repeating the tests all the time is not viable, which makes refactoring and other code changes more likely to result in undetected regression. These automated tests also should be part of a continuous integration framework, so you will know that your whole system works as you include more new code. The process is highly iterative:

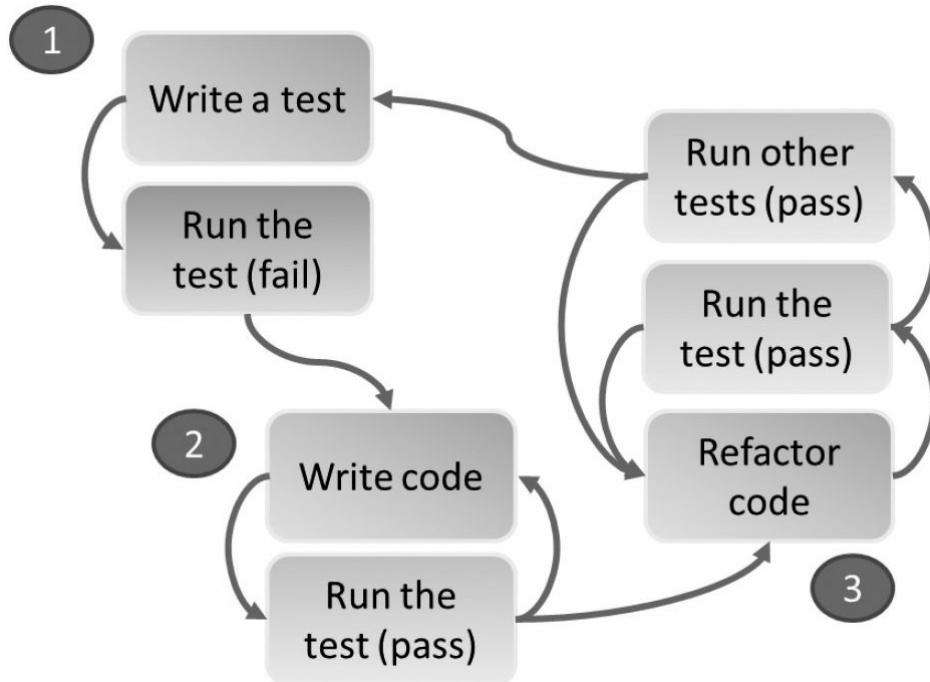
- You write a new test and expect it to fail.
- You write just enough code to pass that test and keep running the test and changing or adding code until the test passes.
- You refactor code for maintainability and run the new test and earlier tests, again repeating the actions if the tests do not pass.

Graphically, this process is shown in [Figure 3.1](#).

The tests you create in this way are unit tests for the code. They can even work as a replacement of some of the technical documentation or technical design. Typically, you would try to achieve at least 100 per cent decision (branch) coverage with unit tests. You can also apply TDD to integration tests and systems tests, although this is seldom done in practice.

---

**Figure 3.1 Process of creating tests and code with TDD**



## Example

Susan is a developer of an e-commerce store's software. Currently she wants to create a postal address input field and avoid mistakes with the input field. She writes a sequence of unit tests that exercise the input field utilising equivalence partitioning and boundary value analysis techniques.<sup>6</sup> As she does this, she also adds equivalent input field validation codes to her code.

She works in TDD fashion. Therefore, she only scripts her first unit test to begin with.

'Try a regular value for the postal address field.' Test should fail, as the field is not coded yet.

She runs the test and it fails, giving a different expected value than was scripted.

Test fails, not surprisingly.

Then she creates the postal address field and adds the equivalent validation check to the code.

'Regular value is allowed.'

She then runs the same test again.

Now the test passes, as the field is there and the validation check also works.

Then she scripts the next test, and adds some more validation code, and so on. In the end, there is a well-validated postal address input field in the user interface.

## Acceptance test-driven development

When doing ATDD as part of an Agile practice, tests are created in an iterative way, starting before, and continuing during, the implementation of a user story (see more about user stories in [Section 1.2](#)).

User stories need to include acceptance criteria and those in turn can be turned into (drafts of) acceptance tests. All this happens through business representatives, developers and testers working together in a specification workshop (Adzic, 2009). The workshop is not only about creating acceptance tests. The real goal is to collaboratively understand what the software should and should not do. ATDD strives to encourage and improve the communication between the business, the developers and the testers. Acceptance tests are a by-product, or, if you will, an exposition of this discussion. The (drafts of) acceptance tests should next be detailed enough that code can be exercised against them. In other words, they tend to be more towards the concrete test end of the logical test–concrete test spectrum. They could be, and usually are, also automated for ease of use. The acceptance tests are continuously refined with additional details about how to interpret the user story and how to show whether it satisfies the expressed user needs or not. ATDD tests are higher-level tests than unit tests, but still quite small tests, each exercising one or more acceptance criteria of a user story. They belong to quadrant two of the Agile testing quadrants, which we'll discuss later in this section. The ATDD process is described in detail in [Section 3.3](#).

ATDD makes it possible to:

- test the code quickly at the user story level, system level or acceptance level;
- verify that acceptance criteria are met;
- find higher-level defects early in user story development;
- automate regression test sets, or at least create automatable elements for such sets;

- improve collaboration between all stakeholders.

ATDD as a methodology does not require test automation, but it is usually coupled with a powerful test automation framework that can take input from specification workshops (with the help of the test automation expert). Some acceptance tests need to access databases, some need to access application program interfaces (APIs) and some the user interface, so usually a framework is also needed from a technical point of view. Frameworks can then use multiple test execution tools.

## Example

The project team wants to understand how the registration to a website will work. The user story is just ‘As a new user I want register on the website so I can use it.’ They set up a workshop and quickly find several questions they want answers for:

- In which order should the process go?
- What information should be received from the user and in which format?
- How should the user be informed about the progress and status of registration?

They take the approach of talking about example answers to the questions (simplified):

- Process example: A user called John Davis could register first his user ID JohnD, then give a password, then give more required information, and then voluntary information.
- Information example: We want to capture, at least, first name (John), surname (Davis), user ID (JohnD), password (#<sup>a</sup>2"dd), email ([JohnD@email.com](mailto:JohnD@email.com)) and country (UK). Voluntary information would be the rest of the address and phone number. The format should be plain text in text fields, with copy–paste and auto-fill allowed. Length limits should be enforced on fields; for example, a maximum of 50 letters for surname.
- Progress example: John should expect to see the registration steps on a progress bar on the top window at all times.

Talking through examples with real data, the team end up adding acceptance criteria to user stories, formatting them as such:

User ID and password must be created first

And also a similar set of acceptance tests:

If user John Davis tries to register user ID John and, having checked the user ID list it is already

taken, he must try again with JohnD, which is then created. Then he is asked for his password.

All this happens during the same meeting, which is the specification workshop.

## Behaviour-driven development

BDD starts from a point of view that behaviours of software are easier for stakeholders to understand when taking part in test creation than the specific tests themselves (Chelimsky et al., 2010). The idea is, together with all team members, business representatives and possibly even customers, to define the behaviours of the software. This definition can happen in a specification workshop, very much like ATDD. Tests are then based on the expected behaviours, and the developer runs these tests all the time as she develops the code.

BDD promotes immediate test automation even more than ATDD. One of the leading ideas is to use very clear English (or another spoken language), so that both business users participating in a workshop and a test execution tool can understand the behaviour and the test that verifies that behaviour. Some test automation frameworks are adapted (such as Robot Framework) and some even created (such as Cucumber and JBehave) to work specifically with BDD methodology.

Participants of the specification workshop are instructed to think how a system will behave. One format is the Given-When-Then syntax, part of the Gherkin format of the Cucumber tool family:

- given some initial context;
- when an event occurs;
- then ensure some outcomes.

The format should also include data that can be used in a test.

### Example

A successful system login scenario should work as follows when described in Given-When-Then format:

*Given* that the system is in the main page or login page

*When* user IDs (Steve, Stan, Debbie) *And* passwords (45KE3oo%&, DF44&aa, 23##a&SK) are inputted to the user ID *And* password fields

*Then* the system will give the message ‘Login Accepted’ *And* move to landing page.

BDD creates business level or acceptance test level tests that can be used by a developer as the team’s acceptance tests or even as part of unit tests. Such tests would be in addition to other unit tests the developer creates as she codes the program, aiming for good coverage (for example, 100 per cent decision coverage). The test automation framework will give the drafts of tests for developers and testers directly. The drafts can then be refined to executable test scripts.

In both BDD and ATDD, the automated BDD or ATDD tests are usually included in the continuous integration framework. This is done to:

- expand the smoke test (or regression test) set to include a business perspective as well as the technical low-level unit test perspective (for example, provided by TDD);
- verify the outcomes and behaviours do not unexpectedly change after a change to the code;
- find defects as close as possible to the moment of introduction, and to fix them upon discovery.

Possible downsides of running the tests in continuous integration include:

- building a new executable might take too long with all those tests included (but you can of course only choose part of the unit tests, ATDD or BDD tests to run each time);
- maintaining tests might become time-consuming.

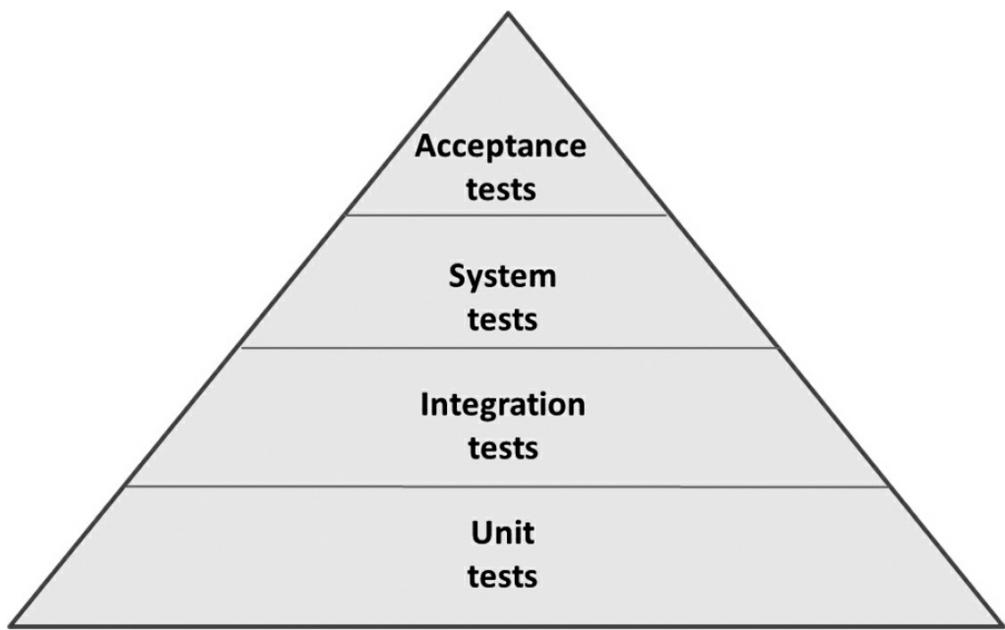
### **3.1.2 The test pyramid**

The concept of the test pyramid was introduced by Mike Cohn (Cohn, 2005), originally with three levels: unit, service and user interface (UI), but we refer to Janet Gregory and Lisa Crispin's popularisation of the concept here (Gregory and Crispin, 2014). Often the test pyramid is referred to as the test automation pyramid, as the basic idea is that all test levels in the pyramid can be automated. Manual or automated, the shape of the pyramid refers to the fact that there are more tests at the lower levels, thus ensuring earlier testing and cheaper defect removal. [Figure 3.2](#) shows the test pyramid with typical test levels.

There will be a lot of unit tests, comparatively fewer integration tests, and so on the higher we are on the test levels. Finally, acceptance tests are relatively few in number. In the end, the test pyramid is a metaphor. For example, while teams should try to test more at the unit level than at the acceptance level, there is no mathematical proportion of acceptance test cases to unit test cases. It all depends on the structure and complexity of the software. Sometimes you might have more integration tests than unit tests due to the nature of a highly complex system of systems. Sometimes you might not have test cases on some level at all. Furthermore, comparing the number of test cases at different levels is meaningless, since the test cases are of different sizes. For example, consider the amount of the system covered by a single use case test at the acceptance test level compared to a single TDD test that might cover just one branch of code in a method (Crispin and Gregory, 2008; Gregory and Crispin, 2014).

---

**Figure 3.2 The test pyramid with typical test levels**

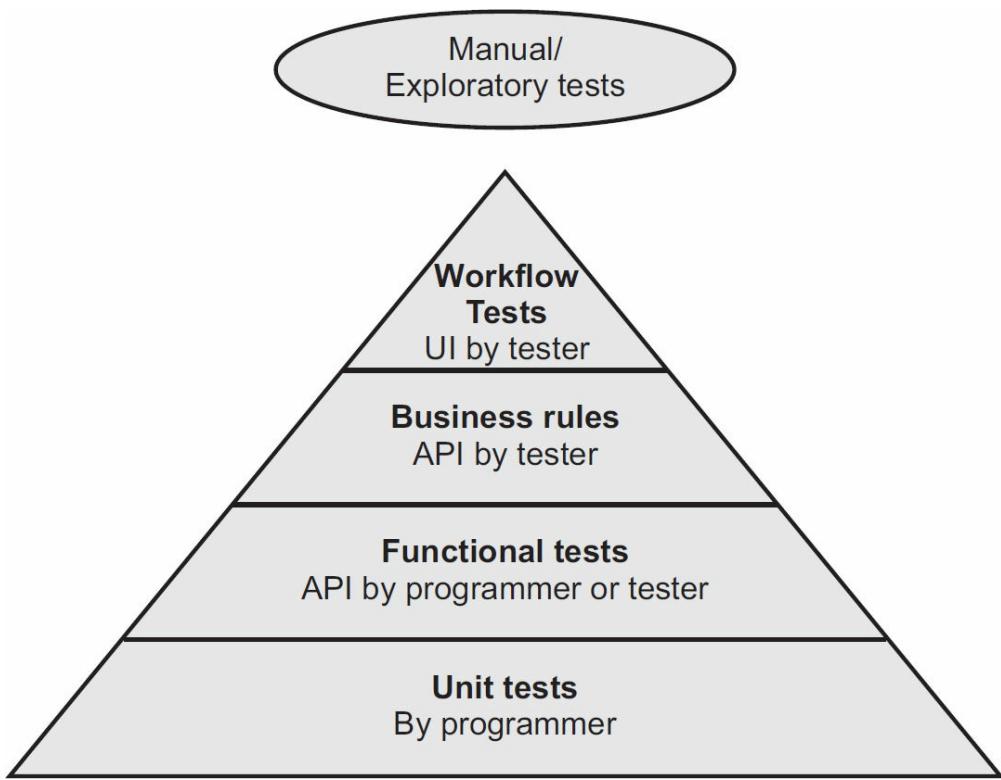


Usually unit and integration tests are automated utilising different API-testing tools, such as unit test frameworks, service oriented architecture (SOA) testers and so forth. System and acceptance tests are typically automated with some GUI-based testing tools or frameworks.

Janet Gregory and Lisa Crispin like to emphasise that, even if all test pyramid levels could be automated, there will still be some manual testing left. This could be exploratory and testing could focus more on validation than on verification. Such tests are very beneficial to the project, because they find defects that other tests might miss. In the context of the test automation pyramid, manual testing can be pictured as a cloud on top (see [Figure 3.3](#)). In addition, the levels could be named differently.

---

**Figure 3.3 The test pyramid with exploratory cloud**



**Note:** This figure is based, with permission, on Gregory and Crispin, 2014: Figure 15.1.

### Example

In an enterprise-level Agile project, called Enterprise, in the finance sector, all developers were required to use CppUnit for unit testing their code. SoapUI was used to test (and automate the testing at the same time) for the functional tests, including more than one unit or component. Both levels of testing were done by developers. The same SoapUI tool was also used to test the more high-level business logic, still accessing the system without the user interface. These tests were done together by a technical tester and a customer representative delegated to the project. Finally, a minimal number of acceptance tests, using the actual workflow in the user interface, were executed, and a subset of those were automated with QuickTest Pro as a regression test set.

### 3.1.3 Testing quadrants, test levels and testing types

If you had to choose one concept only to describe what testing is like in Agile software development, it would be the Agile testing quadrants, or just testing quadrants for short. The concept of testing quadrants was invented by Brian

Marick, and popularised by Janet Gregory and Lisa Crispin (2014). Testing quadrants transfer the traditional test levels and testing types into an Agile context, giving reasons for the existence of various different types of testing.

As seen in [Figure 3.4](#), there are four Agile testing quadrants:

- Q1, which is Technology-Facing and Supporting the Team;
- Q2, which is Business-Facing and Supporting the Team;
- Q3, which is Business-Facing and Critiquing the Product;
- Q4, which is Technology-Facing and Critiquing the Product.

On the left half of the figure are tests that are Supporting the Team. Gregory and Crispin (2014) have renamed these tests as tests that Guide Development, which is much more fitting. As the ISTQB® Agile Tester syllabus (and equivalent exam) currently refer to terms in the previous version of Crispin and Gregory's picture (2008), we are also going to do so to provide consistency. These tests can be seen as verification tests, since they check whether we are building what we defined. These are very much needed in the quick cycle of Agile development. When you add some new, small piece of code into continuous integration, you want to reduce the risk that you may have broken something. Like all regression tests, these tests should not find defects most of the time. They should usually pass and let you know that you are on the right track.

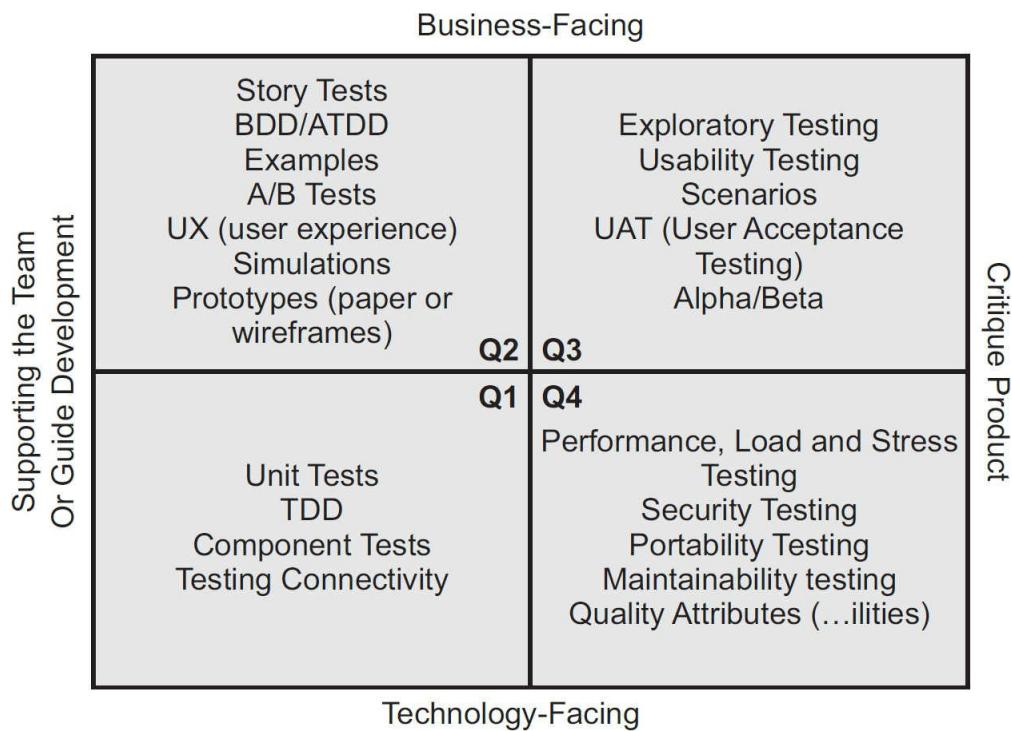
On the right half, you have tests that Critique the Product. These can mostly be seen as validation tests, since they check whether we are building what was needed. These tests are crucial for finding those defects that certainly lurk in any software. Many Agile teams first start with the tests that Support the Team, only to realise that they also need tests that Critique the Product.

On the bottom horizontal axis are the Technology-Facing tests. These are the technical tests that are usually and most easily done by developers or other

technical people. They are often structural in nature, i.e. white-box tests.

On the top horizontal axis you have Business-Facing tests. These are often behavioural in nature, i.e. black-box tests. You need these also. To put it in simple terms, you cannot do unit testing only, you also need to test from the customer perspective.

**Figure 3.4 Agile testing quadrants**



**Note:** Based, with permission, on Gregory and Crispin, 2014: Figure 5.4.

As previously described, around the two axes we see the four quadrants. They are labelled as Q1, Q2, Q3 and Q4 just for ease of reference. There is no implication of order of execution here, as Gregory and Crispin point out in their book. Let us go through the quarters in detail, one by one.

Quadrant Q1 includes unit tests and components tests, which may be the same thing, depending on the project. In addition, integration and API tests would fit in Q1. These tests are typically automated with a unit test

framework, are done by the developer and may be developed using techniques such as TDD. These tests are technology-facing, rather than low-level technical tests that are usually run with continuous integration.

Quadrant Q2 includes functional tests, examples, story tests, prototypes, simulations and other types of testing that could be described as system level tests. These tests are automated if possible, but also can be run manually. They are designed from the business perspective and may involve the use of techniques such as ATDD and BDD. They support the team (or guide development) in a similar way to the Q1 tests. So, they can always be run when considering whether the right thing has been developed, and also to check whether anything has been broken by new code.

Quadrant Q3 includes exploratory testing, scenario testing, usability testing, user acceptance testing (UAT), alpha and beta testing and similar test types that are good in finding defects and checking whether the system indeed does what the users expect, does not produce nasty side effects and does not do anything else that it should not do. This quadrant can be considered system or acceptance test level. These tests are usually manual tests, as they are both difficult to automate and require user and tester experience and innovation.

Quadrant Q4 includes performance, load, stress and scalability testing, security testing and other types of non-functional testing that Gregory and Crispin call quality attributes or ‘-ility’ testing in their books, meaning testing quality attributes such as maintainability, portability and interoperability, as well as tests for memory management, recovery, data migration and infrastructure. These tests find non-functional defects that are related to the different quality attributes. They are often quite technical in nature and may require the use of additional tools. The tester involved is typically quite technical, being either a developer or a tester specialised in these testing types. These tests are automated as much as possible. This quadrant can be categorised as a system or operational acceptance test level.

Each of the quadrants can be exercised in each iteration of the Agile project, although what is being developed also affects what kind of tests are needed. Testing quadrants consider mostly different types of dynamic testing, leaving most of the static testing to be done as separate tasks of the Agile team. For example, reviews, which are categorised as static testing, are not really considered in the testing quadrants concept, and they are still a very Agile method. Security testing, on the other hand (part of Q4), often involves static analysis, such as checking code for common programming worst practices that create security vulnerabilities.

Agile testing quadrants have become so popular that many people have created their own versions of the quadrants<sup>7</sup> and even challenge people to think in ways other than in terms of quadrants.<sup>8</sup> Often people have created versions of quadrants that fit into their own thinking.

### 3.1.4 The role of a tester

Agile methods work well in many different kinds of lifecycle models, some Agile and some traditional. At the time of writing, the most popular Agile framework appears to be Scrum. So, it is worthwhile to take a look at how the tester fits in to Scrum as an example (van der Aalst and Davis, 2013), which we will do in this section.

#### Teamwork

One of the key concepts of testing in Agile teams is the whole-team approach (see [Chapter 1](#)). The whole-team approach means that everyone in the team works together concerning testing and other matters. In other words, we need to work as a team. Successful teamwork is indeed needed for successful Agile projects.

In [Table 3.1](#), Agile team best practices from both team and tester point of view in a Scrum project is discussed, for example that the team needs to

decide together which team member does what (Linz, 2014), and it is essential to have some people with a testing background in the team for this self-organising to work for testing tasks.

---

**Table 3.1 Different Agile team practices for Scrum teams and testers**

Agile practice	Meaning for the team	Meaning for testing
<b>Cross-functional</b>	The team as a whole should include all skills necessary. Originally this was taken to imply that everyone could work with any task, but the need for specialised roles (such as testers) is now widely recognised.	The team must work together on test strategy, test planning, test specification, test execution, test evaluation and test results reporting. The tester might initiate these tasks.
<b>Self-organising</b>	The team decides together who does what, when and how. There is no external project manager role that assigns tasks to the team members.	Ideally, the team should include one or more persons with a testing background.
<b>Co-located</b>	The team, including the product owner, sit together in same space.	Testers, as part of the team, also sit together with the rest of the team.
<b>Collaborative</b>	Team members work together and with the business, customer or any stakeholder to reach the goals of each sprint.	Testers collaborate continuously with other team members, working together rather than alone on their dedicated tasks.
<b>Empowered</b>	The team makes their own technical decisions, together with the product owner and other teams if needed.	Testing tasks and decisions are valued and done together as a whole team, and not dictated by someone outside the team.
<b>Committed</b>	The team commits to completing a selected number of backlog items with good quality within a sprint.	The tester (and thus the whole team) commits to test against the needs and expectations of users and customers.
<b>Transparent</b>	All progress is visible all of the time to any interested party, by using an Agile task board for example.	Testing tasks are also visible on the Agile task board or any other 'information radiators' used.
<b>Credible</b>	The team must only take on tasks that they have the credibility to accomplish. Otherwise, they risk micromanagement from all stakeholders.	The test strategy, its implementation and its execution must be credible and communicated well. Otherwise, stakeholders will not trust the test results and reported team progress.

<b>Open to feedback</b>	Agile teams learn and evolve continuously by learning from feedback and indeed asking continuously for feedback. Retrospectives are natural events in Scrum that make it possible to learn.	Learning to be better in testing must also be the mindset of all team members, especially the tester.
<b>Resilient</b>	Agile projects must be able to respond to change at all times.	Testing must also adapt to changes and testing must be defined so that changes are easily handled, not feared.

Adopting Scrum requires a new enterprise culture (Schwaber, 2007). You have to make sure that you work in a way that fits Scrum. Your team and enterprise around you must respect the Agile values that make the team work. Teamwork practices are not only for the teams, they are what make Scrum work in the context of the whole company.

## Example

Sam was a tester entering into a Scrum team. He was the first tester in the team. Everyone else had been there for many sprints and had years of software development experience. The team had done a lot of testing, but, as Sam was a testing professional, he soon noticed that more could be done.

Sam decided to utilise teamwork best practices to get some more testing done. He started with the Open to Feedback practice. He brought up a few of his ideas where he thought the team was underperforming. He brought up two items in the first retrospective he attended:

- There is no performance testing.
- Testing from a business point of view is limited to a few automated Selenium tests, running with each continuous integration build.

He considered Credibility to be the key practice, saying,

Look, I have been involved in business-oriented testing and performance testing for many years, and have always found lots of interesting potential improvements. I can show them to you in a few hours.

He then brought up the Cross-functional practice, saying,

We need to do this together, of course. Business-perspective tests and performance testing are both practices that all of us can learn, and I can't become a bottleneck for activities. I might get

ill, or you guys might be very fast with your own tasks.

He moved on to the Committed practice, saying,

I'll take care of these in the first sprint.' He continued with the Self-organising practice, saying, 'Who wants to pair with me in the first sprint to learn these testing activities? We can alternate activities sprint by sprint.

## Sprint Zero

Many teams start their journey with Scrum in Sprint Zero, where they initialise things that are needed further on the journey. Some teams call it Sprint Zero, some just do these actions before the first sprint without calling this set of actions by any name, and some include these tasks in Sprint One. Whatever you call it, there are a number of considerations that teams need go through before working in an Agile fashion, such as setting up a task management system or an Agile task board.

The tester should collaborate with the whole team in Sprint Zero and make sure that test-related items are included in the task list of the sprint, and testing perspectives are taken into account in implementation of the rest of the tasks. [Table 3.2](#) considers these items.

---

**Table 3.2 Typical tasks of Sprint Zero and their implications**

Task in Sprint Zero	Related testing	Relevant testing perspective
<b>Identify the scope of the project</b>	Identify test emphasis based on the scope.	Think about the whole product backlog to understand the scope. Bring testability into the discussion. Ask what would make sense to implement first.
<b>Create an initial system architecture and high-level prototypes</b>	Think about a test automation architecture.	Talk about testability, and make sure the architecture enables it; for example, easy access to test automation tools.
<b>Plan, acquire and install any needed tools</b>	Get tools for test management, defect management and test automation.	Discuss the integration of test cases into task management and the integration of test tools into task management and reporting.

<b>Create an initial test strategy for all test levels</b>	Based on the scope of the project, define what kind of testing is needed in the project and when. Consider (among other topics) test scope, technical risks, test types, test methods, test criteria and coverage goals.	Discuss what testing tasks will be part of other activities; for example, tasks done with each coding task.
<b>Perform an initial quality risk analysis</b>	Think of quality risks to bring up in the risk analysis session with the whole team.	Determine the risk for all backlog items; for example, by using the risk poker method discussed in Section 3.2.
<b>Define test metrics</b>	Measure the test process and the progress of testing in the project.	Decide how to measure product quality.
<b>Specify the definition of done (DoD)</b>	Think of testing tasks you want to accomplish, and try to fit as many as you can into the team's DoD. The remaining tasks might become the tester's own tasks if there is still time.	Decide what type of testing and to what extent (coverage) must be part of the DoD.
<b>Create the task board</b>	In some cases, set up a separate testing task board; for example, for integration testing of several user stories.	Ensure task boards include testing tasks and testing columns, so that testing progress can be seen.
<b>Define when to continue or stop testing before delivering the system to the customer</b>	Define test criteria that would help to decide when to release.	Determine such a set of release criteria that utilises test criteria and other information from testing.

Whatever the team's decisions on testing are, they will influence more than the first sprint. Good decisions are long lasting, and this should be kept in mind. Under Agile principles, every retrospective at the end of each sprint

offers a chance to change any of these decisions and preparations. However, it would be too time-consuming to change everything every sprint. So, think carefully about decisions made in Sprint Zero.

## Example

James was a tester chosen for a new team that would develop a new mobile game. The team decided to have a formal Sprint Zero and have (for once) time to do things right based on their accumulated learning from a number of Agile projects and earlier retrospectives.

The team started talking about the need to acquire a new task management tool. James pitched in to the discussion noting that testing tasks need to be separated from the user story descriptions so that they can be tracked individually. He explained that there are many task management tools that allow this. ‘Indeed,’ he said,

our previous tool also would have worked that way, had we configured it properly. Let’s look at a few that could do this, ok? I know that Jira, VersionOne and TFS can do this, and we can look at others once we have our tool requirements and constraints fully defined.

The team then discussed what kind of testing needed to happen. James pointed out that, in addition to the unit testing and acceptance testing that most developers were very familiar with, they would also need access to real alpha and beta testers from the gaming community.

If we release an alpha to some key gamers after every sprint, we’ll be able to get ideas about what works and what doesn’t. We could even have a feature enabled for some and disabled for others in order to see which option is more attractive to the users, so let’s consider this A/B testing as well.

He then continued to say that he would like to set up a testing page in the wiki they already had.

It will be fast to create, no worries, not much text at all, just to document what we’ve agreed. In fact, I can do most of it right now as we are discussing it. I’ll call the page Test Strategy, ok?

He then switched the signal on his laptop to a screen that they could all see, and they continued their preparations.

## Integration

Scrum aims to deliver a potentially shippable product increment at the end of every sprint. This means that whatever is coded within the sprint must also be tested. Testing should occur on a continuous basis and, for this to happen, integration needs to be continuous. Continuous integration is, of course, the

solution, but it needs sufficient testing environments so that the testing can always work on the latest or almost latest version of the software (depending on how hard it is to keep the integration environment stable). Testing environments in general should be acquired and managed as soon as possible, as creating them is usually a slow process (Gerrard, 2009).

Testing on higher levels needs to have enough corresponding functionality available to make any sense of testing, so the dependencies and relevant quality characteristics should be considered. This should directly influence the implementation priority of different user stories and features.

## Example

Sarah wanted to ensure her project could have a continuous testing strategy. She wanted to be able to test anything that developers produced, at the latest, the next day after the implementation. She viewed the matter in such a way that every day she would have something new to test.

She realised that there would be some challenges before reaching this goal. She also realised that the developers fixed the integration environment for several hours almost every time something was committed into it. The developers said it was the normal practice of unit and integration testing, making the software work by fixing the code and running the automated unit tests over and over. However, as a side effect of this, Sarah had the environment available only a few hours per day.

After some consideration, she proposed in the next retrospective the following changes to the integration strategy:

Let's have another environment for executing manual acceptance tests and exploratory tests.  
Let's have the CI engine refresh this new environment every time, once all the automatic tests have passed in the integration environment. This way we'll always have a stable acceptance test environment.

Let's consider which features and user stories make up the new main functionality of the ongoing sprint. Let's code and integrate them first, so they get to acceptance testing earlier. Can we make this work?

She got positive feedback on these suggestions and the team adopted these practices in the next sprint. Point 1 was addressed via a separate task in the sprint for setting up the new environment. Point 2 was addressed through prioritisation in the next sprint planning.

## Test planning

Planning the test activities is an integral part of planning the whole sprint. Thus, it is present in both release planning and sprint planning. Any release level testing decisions could be updated in sprint planning, but of course most of the test planning within sprint planning is about more detail, about what will be tested and how in the sprint (see [Section 2.1](#) for greater discussion of release and iteration planning).

When a sprint plan is developed, each user story is split into several tasks, some of which will be testing tasks. In addition, independent testing tasks can be put into the sprint backlog and task list. Typically, each task is one to two days' work, but as everything is decided by the team, so is the task length. From a project management best practices perspective, tasks should not be planned to take longer than two days because otherwise they are hard to estimate and often contain hidden subtasks and dependencies that result in delays.

Tasks, including the testing tasks, are then displayed and tracked on the team task board. The task board should provide a clear view of the status of each task. Often, testing tasks are shown in different colours to make them stand out from the rest of the tasks. In addition, defects could be put on the task board if the changes they require are too large to fix immediately.

### Example

Mandy was a tester in an embedded-software team, producing software for internet routers. She had joined the project recently, and had now convinced the others that she should take part in the team's release planning (or grooming, as they called it) session.

She entered the grooming session, where the product owner and two main developers were discussing high-level epics. They considered the user stories within each epic, and the order in which the user stories would be developed, and thus the sprint in which each would be implemented. This led to product backlog prioritisation.

Mandy commented on one epic that if they were going to have the new support for 100-device

nets to replace the existing support of 50-device networks, they would need to invest in and build additional devices into the test lab as well. This comment was well received and a separate task with a dependency to the relevant epic was created. The task was, however, created as a separate backlog item and given a higher priority so the test lab would be ready in time. Ordering those additional devices would take time.

Later on, in the next sprint planning meeting, Mandy was happy to see the Test Lab Expansion backlog item incorporated by the team into the sprint backlog. They even decided to break it down to several tasks due to the considerable time (estimated as a week) required to set up the test lab.

A few sprints later, the new epic for 100-device network support arrived in the sprint backlog. It was so big that it had been split into six user stories, and when the team worked it up, it turned out that only one other minor user story would fit into the next sprint. Each of the user stories was divided into tasks, one of which was always the testing task. She would then work with the test management system to list the actual test cases and areas she wanted to test related to each testing task.

## Pairing

One of the useful Agile practices in an Agile team is pairing up. It might be classic pair work with only one keyboard, with people alternating using the keyboard. Alternatively, it might be people sitting next to each other, sharing the same task and discussing it all the time. For developers, this usually means pair programming, but for testers there are a number of attractive options. They are considered in [Table 3.3](#).

---

**Table 3.3 Different testing-related pairs in an Agile team**

<b>Pair</b>	<b>First member of the pair</b>	<b>Second member of the pair</b>
<b>Developer–Tester</b>	Developer implements code and unit tests, tries to think aloud and listens to the feedback of the tester.	Tester sits next to developer and engages in continuous discussion with the intention to clarify the functionality being developed to both the developer and herself. She can also write some test cases at the same time, for further higher-level testing on the same functionalities. She may also help in finding the location of the defect; i.e. in the debugging process.
<b>Tester–Tester</b>	First tester is using the keyboard and navigating around the application; for example, doing exploratory testing.	Second tester observes the first tester, comments on what to try next and can also log what they are testing. At intervals, she can switch places with the first tester.
<b>Tester–Product Owner</b>	Tester brings in her testing knowledge, and designs and executes tests.	The product owner (or other business analyst or other business representative) brings in her business/domain knowledge, discusses with the tester what would be beneficial to test to get more coverage and explains how users will expect different functions to work.

## Example

Sheldon works in a Scrum team that develops a pension insurance system. Carrie is the product owner of the same team. Sheldon has been to a pension insurance course to understand the key concepts of how pensions are calculated, but he still considers himself a novice in pension insurance. Carrie, however, has 20 years of experience in managing the systems within the pension insurance domain.

They decide to pair up and work together to create a superb test case set for the ongoing sprint and to execute that test case set. They start from the priority 1 back-log item that the developers have started to implement on the first day and continue in the same priority order that the developers use to code and unit test the user stories.

During the first day, they are so successful in their test design that they go through 10 user stories and create test cases for them. They only create the name of the test case (reflecting the purpose of

the test, which is to say, the test conditions to cover), not bothering to create detailed test steps as they will test them together. This is good, as the next day they get implemented code of the first user story from the developers and spend the entire day manually executing their tests for that user story, with Sheldon driving the computer and Carrie commenting and making notes. They decide the detailed test steps as they go forward with the test execution.

The first user story is ‘log in into the system’. Their test case list is just:

- test with valid login ID and password
- test with several invalid login IDs and passwords
- try to bypass login page into the main page

The day after, the next user story arrives, and they need to start testing that functionality. They stop to consider what this means for their work, as they had just one day to execute tests for the first user story. They agree that together they can keep up the pace of the developers, but Sheldon alone probably would soon become a bottleneck for the rest of the team.

## Incremental test design

Scrum typically works in one- to four-week sprints, in which time tests also need to be designed and executed. In the same way as the code is developed incrementally, the test cases and test charters are also developed incrementally to keep to this schedule. Indeed, with test charters in exploratory testing, you move very quickly when executing the tests and deciding on the next test case based on the previous test result and what you learned from it.

User stories and other test bases are used as a background for test design, but tests can be based on any information available. Incremental design usually starts from simpler tests, followed by more complex ones.

### Example

Larry has decided to engage in exploratory testing in his Scrum project. He has been an integral part of all release planning and sprint planning discussions, and has read all the details in the user stories.

On the first day of the sprint he goes through the user stories and builds up high-level test charters,

mapping all software areas that he probably needs to touch in his testing. He spends just half an hour per user story, and is just finished when the first new piece of code comes into the testing environment late on the first day.

He takes a look at what actually arrived by looking at the user story and the software now running in the test environment. He makes a few changes in his test charter to note which areas to be tested have been logged and adds one missing one. Then he starts testing using the exploratory testing approach.

As that day and the next day goes by, he proceeds through his test charters (over several test sessions), designing more detailed tests on the fly, when he sees the need for more precise coverage than just trying some values out. At the end of the day his test charters and test logs include a lot more test designs as well as the test results.

Then it is time for the next user story. He repeats the same pattern, making adjustments on how much time he spends on each task, and goes on finding defects.

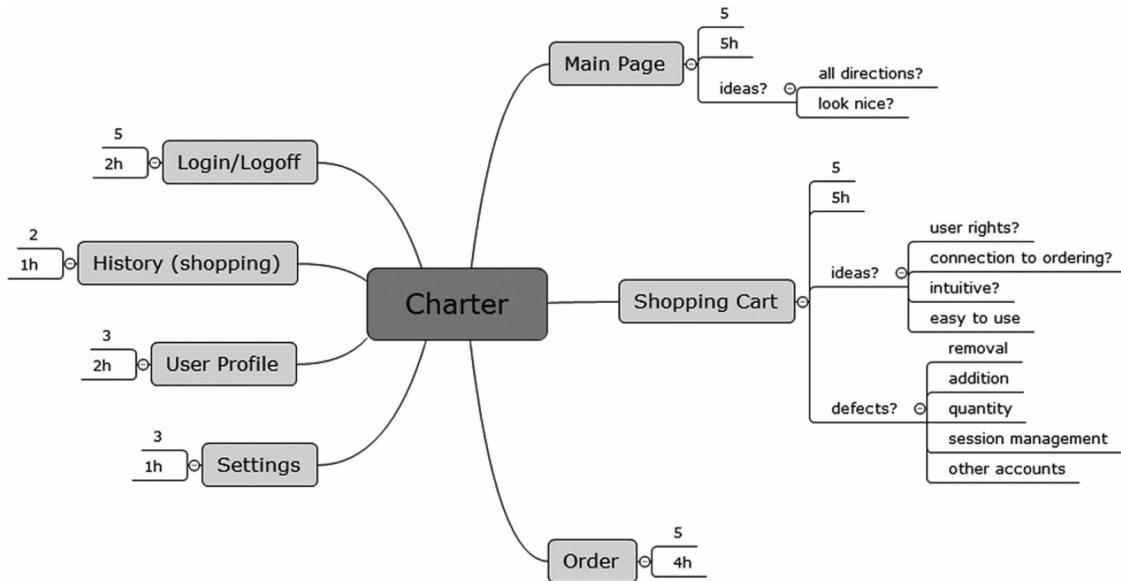
## Mind mapping

Mind mapping shows lots of promise for, and compatibility with, the Agile way of working. Indeed, it has been adopted by many Agile testers. Mind maps can be used for almost anything, say the enthusiasts. It is quite natural to plan your exploratory tests with mind maps (test charters and test logs), to create the test strategies or test plans with them or to describe test data.

Mind maps are strong when you need to enter data quickly and categorise it in a meaningful way as you go. Moving items around (with a mind map tool) is a quick drag-and-drop activity. A range of mind mapping tools from both the open-source and commercial worlds are now available. You can find an example of an exploratory test charter done using a mind map in [Figure 3.5](#).

---

**Figure 3.5 An example of a simple test charter done with mind mapping tool Xmind**



## Summary

In this section, we have discussed Agile methods that help in the work of the Agile project. We have discussed the test-first approaches of TDD, ATDD and BDD. We have looked into concepts such as the test (automation) pyramid and Agile testing quadrants. We have gone through some practices that help you to understand the role of the tester in Agile projects, and in Scrum specifically. Not everyone uses all the methods, practices and tools available, but most teams would adopt many of them to solve the testing challenges of an Agile project.

## Test your knowledge

The answers to these questions can be found in the Appendix.

### Agile Tester Sample Questions – [Chapter 3 – Section 3.1](#)

#### [Question 1 FA-3.1.1 \(K1\)](#)

Which of these is used primarily by a developer?

**Answer set:**

- A. Test-driven development
- B. Behaviour-driven development
- C. Domain analysis
- D. Acceptance test-driven development

### **Question 2 FA-3.1.2 (K1)**

Which of the following statements is true concerning the test pyramid?

**Answer set:**

- A. Unit tests can be automated with GUI-based tools
- B. All levels of the pyramid could be automated
- C. Acceptance testing is always done manually
- D. It is cheaper to find defects on higher levels of the test pyramid

### **Question 3 FA-3.1.3 (K2)**

Which of the following is true?

**Answer set:**

- A. Usability tests critique the product and functional tests are business facing
- B. Unit tests are technology facing and performance tests are business facing
- C. Usability tests support the team and user acceptance tests are business facing
- D. Unit tests critique the product and load tests are technology facing

### **Question 4 FA-3.1.4 (K3)**

A sprint planning meeting for the first sprint in a Scrum project is about to start. You are the tester in the team and want to make good use of both your time and everyone else's on the first sprint. What would you suggest in the meeting?

**Answer set:**

- A. All team members should adopt mind mapping as it is such a powerful technique
- B. The team should include preparation tasks in the first sprint and commit to fewer user stories
- C. Developers should plan the development tasks and let you plan the testing tasks
- D. Continuous testing should be adopted instead of continuous integration

**Agile Tester Exercises – Chapter 3 – Section 3.1**

**Exercise 1 FA-3.1.4 (K3)**

You are the new tester in a Scrum team. The team is also quite new, and altogether comprises seven people. Most team members have been in the team for only three or four sprints, with two developers remaining from the original team that started less than a year ago. There has not been anyone in the team so far with a testing background. The developers have done extensive unit testing and automated those tests as well. They are able to create a new build every two days. The project has released one version of the software to users, after six months. It was not a very successful release, resulting in many reports of failures from the customers. Several team members were changed, and more focus was put on testing.

What will you choose, as a few guiding principles,

1. for testing in an Agile project?
2. for your own role in it?

## 3.2 ASSESSING QUALITY RISKS AND ESTIMATING TEST EFFORT

The learning objectives for this section are:

FA-3.2.1 (K3) Assess quality risks within an Agile project

FA-3.2.2 (K3) Estimate testing effort based on iteration content and quality risks

This section first covers risk-based testing in Agile projects. We start by asking a basic testing question: When is software ready to release? Then we consider software risk in general. We explain and justify why risk-based testing is necessary, and how to test riskier user stories.

In the second part of this section we deal with assessing quality risks in Agile projects. The key part is risk poker, which is a great tool for predicting quality risks in an Agile project. We show how to visualise risks and give examples on how to apply risk poker in practice. Finally, we map a set of testing methods to be applied for user stories at different risk levels.

In the last part of this section we play planning poker, a method that estimates the necessary effort for implementing user stories. We also show how to apply these pokers together.

### Risk-based testing in Agile projects

We are not living in an ideal world. Nothing is perfect. There is a limit to the precision of every instrument. Software is no different. Developers make mistakes. The tester's role is to create tests that reveal the results of these mistakes, the defects in the code, by causing failures to occur. Ideally, we would test every possible input; however, that is impossible. Even for very

simple code containing only two 32-bit integers, the number of possible test cases is  $2^{64}$ . Assuming each test execution takes one micro second, the total testing time would be more than  $10^{44}$  seconds. That is more than  $10^{36}$  years, which exceeds the current age of the universe.

If exhaustive testing is not possible, perhaps there is some general method that can provide complete confidence in our software? Such a set of tests must contain at least one test that fails if the software contains a defect, and all tests would only pass if the software were entirely defect-free. Unfortunately, it has been proven that no such set of tests can be generated for any but the simplest software.<sup>9</sup> That is why test design is so important and challenging. Given the omnipresent push to accelerate time to market, the time for testing is limited. In Agile projects, the use of test-first methods can reduce the well-known squeeze on testing. Developers should avoid late delivery of previously untested software to testers. Instead, the whole Agile team has to optimise every step of the software development lifecycle, including testing activities, throughout the iteration.

Since the number of possible tests is practically infinite, and no set of tests that gives complete confidence in the absence of bugs can be created, a frequently asked question about testing, both in traditional and Agile projects is: How much testing must be done?

Let us start with a simple example, where all the different parts of the software are tested in the same way and with the same amount of effort. The bugs are entered in a bug tracking system, and the developers try to fix the bugs. The blocking bugs are fixed first, then the critical ones, and so forth. In most such cases there will not be enough time to fix all the bugs; therefore, the project leader must decide the order of priority in which bugs should be fixed.

In this situation, the most important question will be when the software can be released. The answer depends on many factors. It is not enough to

consider the software alone. We have to consider the alternatives. For example, 20 years ago GUI-based operating systems frequently failed so severely that they had to be reinstalled. Installation was difficult and required some IT knowledge. At that time, these operating systems were sellable since there was no better option and using them was preferable – at least to many users – than the non-GUI alternative operating systems. Nowadays nobody would use such failure-prone software.

From one pragmatic point of view, a particular piece of software is acceptable if the users of the software get some significant advantage when using it compared to the use of nothing or another piece of software that serves the same purpose. This criterion permits the release of software with bugs, provided those bugs will not cause failures severe enough that users would prefer using another piece of software, or nothing at all. This gives us an important possibility: we do not need to test everything, but rather should focus on reducing the risk of unacceptable behaviour, so that the software under test meets the criterion above.

This criterion, though, does not give us any way to decide whether our software is acceptable. It only gives us a negative statement: the software cannot be worse than any other alternative software or worse than the alternative of using nothing. At least this criterion frees us from the idea of testing everything or striving to release defect-free software, and points us towards a solution.

Rather than test everything the same way and to the same extent, we can look for a way to optimise testing to minimise the risk of unacceptable behaviour. We can classify the user stories with respect to the amount of testing required to meet this criterion. The method for this classification is quality risk analysis.

So, let us take stock of where we are. We know it is impossible to test exhaustively. We know we cannot create a set of tests that will give complete

confidence that the software will not fail in use. Further, we know that we must work within available budget, time and resource constraints. This gives us three choices in terms of the amount of testing assigned to each user story:

- Make no deliberate decision in terms of user story testing, and proceed aimlessly until the constraints require that testing stops.
- Test every user story with the same amount of effort and priority.
- Make a careful decision about the priority and amount of effort for testing each user story.

It is clear from reading the alternatives that the third option is the most common-sense. The most efficient and reliable way to implement the third option is risk-based testing.

## **What is risk?**

Risk can be defined as the possibility of a negative or undesirable outcome or event. The level of risk is determined by two factors:

1. If the impact or damage is larger, then we are facing higher risk; a car crash on the highway poses a higher risk than a small collision when entering a parking place, for example.
2. The higher the probability of a negative event, the higher the risk; for example, that is why driving a car is more risky than travelling on a train.

Based on these factors, the level of risk is the impact caused if the bad event could occur multiplied by the probability of the bad event occurring.

We can apply risk analysis to stories. A story risk is the level of risk associated with that story failing; i.e. the impact of the failure multiplied by the probability of failure. If there is the high probability of a failure, then this failure will happen during the lifecycle. If the impact of the failure is severe,

then either we fix the defect prior to release or it will cause unacceptable behaviour for the users. This leads to another common-sense conclusion. Where the story risk is higher, we should test more and earlier, and where the story risk is lower, we should test less and later.

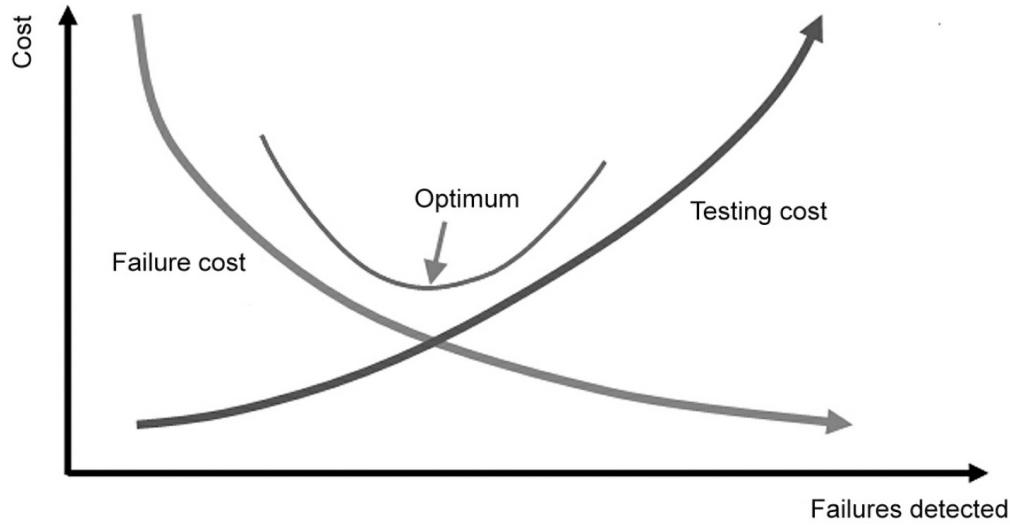
### **Why risk-based testing is necessary**

Testing effort increases exponentially if we strive to execute more and more tests by applying stronger testing criteria. The cause is the combinatorial explosion described by Boris Beizer (2008). Suppose we have 10 input fields on a screen, each with four equivalence partitions. Assuming complete independence between the fields, and all partitions being valid to cover the partitions, the concept of equivalence partitioning testing (as described in the Foundation Level syllabus) specified that we need only four tests, because we do not worry about combinations of fields. However, if we must cover all pairs of partitions across all possible pairs of fields, we need at least 16 tests, or  $4 \times 4$ . To cover all triples requires 64 tests, or  $4 \times 4 \times 4$ . To cover all combinations of partitions requires 1,048,576 tests, or  $4^{10}$ .

The impact of failure is also exponential. This means that, if testing is weak and most of the bugs are found in production, the cost of fixing them increases exponentially. Based on this, an optimum can be reached, as shown in [Figure 3.6](#) (in [Figures 3.6–3.8](#), we use the term failure cost instead of impact of failure to be compatible with testing cost).

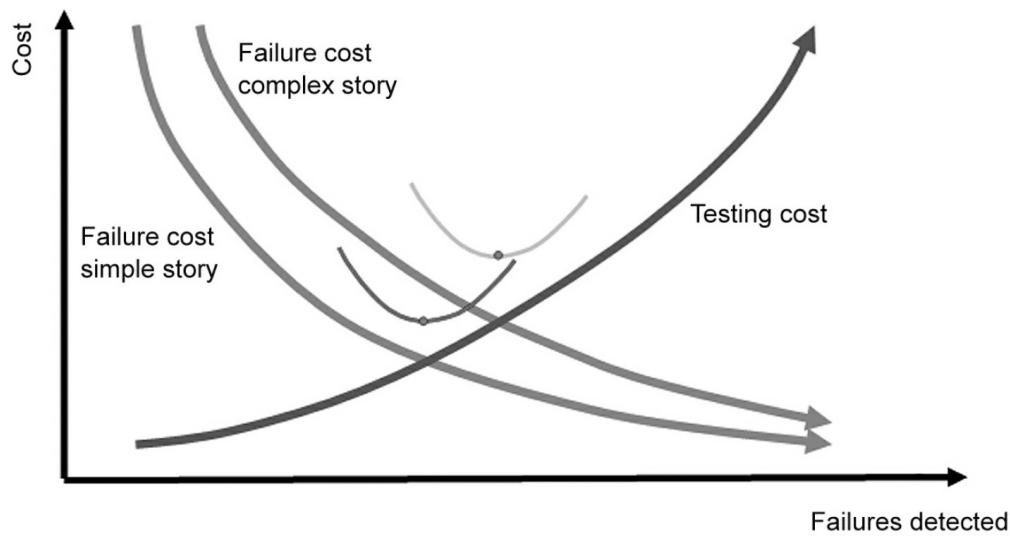
---

**Figure 3.6 Optimising failure cost**



For different user stories, the failure cost is different. Let us assume two stories, where their size is the same but the second is more complex (and thus more likely to contain bugs). Considering the same testing cost, for the second story the failure cost is probably higher, because there may be more failures remaining in the code. Illustrating the failure costs for both simple and complex stories, as in [Figure 3.7](#), we realise that more testing effort is needed for complex stories to reach the optimum. Note that the increased testing effort is associated with the higher complexity.

**Figure 3.7 Failure costs differ for each story**

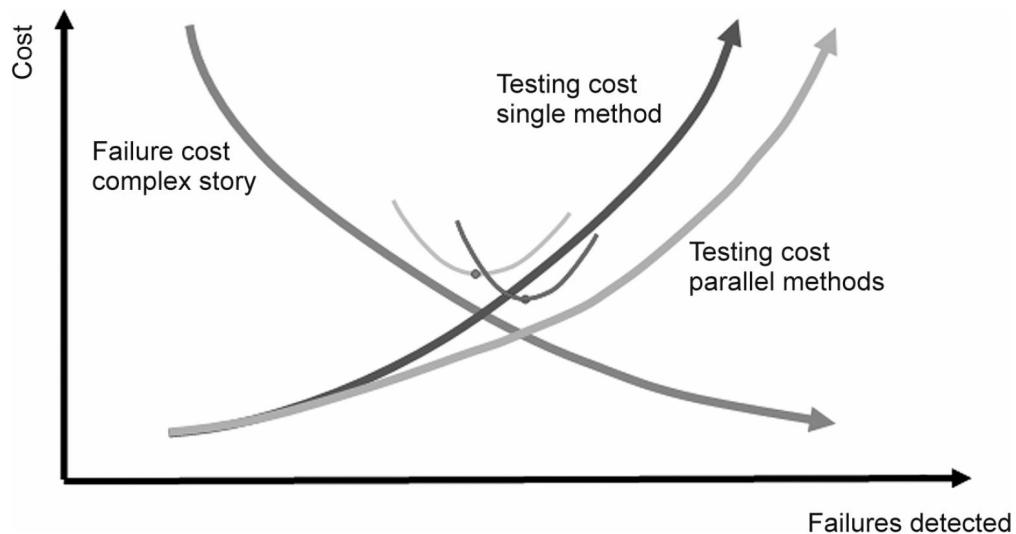


That is the reason why risk-based testing is applied. If a story is more risky – either due to a higher probability of bugs, a higher criticality of the story, or both – then stronger tests need to be applied. The risks can be analysed and the necessary testing priority and effort can be determined prior to coding.

To avoid combinatorial explosions, we can use different testing methods in parallel rather than simply blindly trying to increase the percentage of potential tests we run. For example, we can apply exploratory testing, equivalence partitioning with boundary value testing, defect prevention (through early test design such as ATDD and BDD) and static analysis together. In this case, the testing cost with respect to the detected failures does not increase exponentially. The reason is that different methods will detect different sets of bugs, though the sets do partially overlap. Now the cost curves look like [Figure 3.8](#).

---

**Figure 3.8 Optimising failure costs with multiple test methods**



The figures show us that risk-based testing guides not only the determination of test priority and allocation of test effort, but also the selection of test design techniques. For riskier user stories, we must not only use stronger coverage criteria, but we must also apply a wider range of test design

techniques.<sup>10</sup> For less-risky user stories, we can use weaker coverage criteria and fewer test design techniques.

Now we can summarise why and how risk-based testing is applied. Test techniques vary in thoroughness of testing. Furthermore, for many techniques, the thoroughness of the technique can be adjusted through the coverage criteria used for the technique. For every user story, there is an optimum selection of test techniques (including the coverage criteria for each technique) that will minimise overall cost, as shown in [Figures 3.6–3.8](#). Our aim is to get as close as possible to this optimum through a careful selection of test techniques and their associated coverage criteria. Applying risk analysis, our experience and metrics on previously observed failures, we can select the best test techniques and coverage criteria for a user story. As we will see later in the discussion on risk poker, in some cases the risk can exceed a limit, making it impossible to figure out the optimum set of test techniques. In this case, we must modify the user story by breaking it into simpler stories.

We will return to the topic of test design techniques later in this section, but first we will discuss practical ways to determine the level of risk based on failure impact and likelihood.

## **Impact of failures**

Let us focus first on impact (or damage), which is one of the factors determining risk. For example, we can use the four-level scales that are used in ISO and IEEE standards. These failure severity hierarchies are:

- catastrophic;
- critical;
- marginal;
- negligible.

A failure is **catastrophic** if the software stops working entirely, especially if it causes damage to people or the environment, stops other programs or crashes the entire host system. The failure is also catastrophic if important data is utterly lost. The most serious impact occurs when large financial losses, death or injury results.

### Example

One of the most famous examples is the catastrophe of *Ariane 5* in 1996. The total damage was US\$370,000,000–500,000,000.

The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system. (From the report by the Inquiry Board: <https://ima.umn.edu/~arnold/disasters/ariane5rep.xhtml>)

A failure is **critical** if one of the key features does not work or works badly, especially if no workaround is possible to restore the missing or erroneous functionality. A failure is also critical if a failure in a very important feature happens frequently, and only a very time-consuming workaround resolves the problem. Another case occurs when some data is lost or corrupted and restoring the data is very time-consuming.

### Example

If the *find* feature in a text editor does not work at all.

If the *find* feature in a text editor goes wrong after the first usage, and a restart has to be done for it to work again.

A failure is **marginal** if the user can solve the problem by a fairly quick and simple workaround. A failure is also marginal if some data is lost, but can be restored easily.

## **Example**

The text editor crashes sometimes; however, it makes a back-up so that very little text should be rewritten and the time loss is less than 1–2 minutes.

A failure is **negligible** if the problem does not affect functionality, but rather makes the product less appealing to the user.

## **Example**

Starting an application takes too much time prior to the first screen appearing.

This is a general impact classification, which has to be adjusted for the team's local conditions and requirements.

We have considered the impact of a single failure when a single user applies the software. The total impact depends on how often a failure happens during the overall usage of the software. We can also classify the frequency of usage:

- always;
- frequent;
- occasional;
- rare.

The total impact is determined both by the failure severity and the frequency of usage of this feature.

## **Always**

Those parts of the code that are (almost) always executed after the software is started. In the case of servers: where the application is continuously running those parts that are inevitably used, such as save the modifications, login

periodically and so forth. For example, the spell check function in a text editor is always used (or at least it should be used for everybody).

## **Frequent**

Parts of the product code that most users are using frequently, but maybe not during every usage session. For example, a search function in a text editor is frequently used.

## **Occasional**

An area of the code that an average user may never visit, but is needed for a more serious or experienced user occasionally. For example, inserting a hyperlink in a text editor is used occasionally.

## **Rare**

An area of the code that most users never use, and which is used only if customers do very unusual activities. For example, inserting a special ‘e’ character is very rare, though this author has done it when writing this chapter.

## **Probability of defects**

The other factor in determining the level of risk is the probability of defects (also referred to as the likelihood of defects). During quality risk analysis, we should predict where we expect the code to contain many, moderate, few or almost no defects. What should we consider when trying to predict the probability of defects?

## **Complexity**

Complexity is one of the biggest sources of defects. The more technical or business aspects the programmer must take into account when programming,

the greater the chance of introducing a defect. The same is true for system architecture, database design, network design and all other parts of building a system. While various metrics exist to measure the complexity of developed code, in quality risk analysis we have to estimate complexity prior to coding. To do so, we can consider the user story, which should describe the desired behaviour of the software. If a story is complex and difficult to understand, we would expect that the related code will also be complex. Implementing a very difficult algorithm will be complex, too, and we can expect more defects in it.

### **New technology, methods and tools**

Developers who start using new technologies, methods or tools are at the beginning of a learning curve. They inevitably make many more errors than they will once they have reached the top of this learning curve. For example, consider the programmer using a new programming language. He or she must consider the language syntax and semantics as well as the story to be coded. Sometimes the semantics of the language will differ from what the programmer is used to, leading to a different operation from expected. Using new technologies may be difficult, especially when they differ from what the programmer has used before. For example, programmers writing their first mobile app may not be aware of how central processing unit (CPU), memory and storage limits differ significantly from PC applications. Even moving from traditional lifecycles to Agile methods – learning to apply test-first methods, for example – can result in more mistakes at the beginning.

### **Other**

There are some other factors that makes code error-prone:

- **Changed code:** Changes may unintentionally affect other parts of the code, leading to the introduction of severe defects. It is almost

impossible to know all the ways that one piece of code can interact with other pieces of code, especially for very large programs. Another problem with changing code is that, except when refactoring, the code quality often degrades.

- Time pressure: If the deadline approaches and many stories remain to be completed in the iteration, then the quality of the code will decrease.
- New team members: For newcomers, everything is new and they will be at the beginning of a learning curve.

Besides these general factors, other ones may also be considered, such as geographical distribution of the team, an overall lack of experience in the team, and so forth.

### **3.2.1 Assessing quality risks in Agile projects**

We have explained the value of risk-based testing, and the ways that impact and likelihood affect the level of risk for any given user story. Now we need a method to analyse the quality risks for user stories. Here, we will look at two methods. The first is an adaption of typical lightweight quality risk analysis techniques. The second is an interesting twist on these well-established quality risk analysis techniques, where you adapt an Agile estimation technique (itself derived from the Delphic Oracle technique<sup>11</sup>) to quality risk analysis.

#### **Iterative lightweight quality risk analysis**

The process of quality risk analysis in Agile lifecycles is a bit different from what happens in a sequential lifecycle model. In a sequential lifecycle model, the entire quality risk analysis occurs at the beginning of the project, ideally in parallel with the development and finalisation of the requirement specifications. That initial risk analysis is periodically adjusted at major

project milestones.

In Agile, as you can imagine, that process is turned on its head. There is a high-level quality risk analysis done with business stakeholders, technical stakeholders and the testers during release planning. The discussion is about what worries the business stakeholders from a quality point of view. There is no point in being specific, because the content of the system can change. The specific quality risks are identified and assessed during iteration planning. That process involves the business stakeholders, technical stakeholders and developers.

Once quality risk analysis is complete for the iteration, the design, implementation and execution of tests for that iteration follow. The quality risk analysis that occurs during release planning is not followed by the design or implementation of tests. Some find that problematic. Is there enough time in each iteration to address the relevant risks, especially non-functional risks related to performance and reliability? Sometimes the answer is no, so some organisations handle those kinds of risk items outside the context of the iteration teams.

Now, other than the iterative nature of the quality risk analysis, the process is basically the same as what happens in sequential lifecycles. The whole team evaluates the iteration. The team identifies the specific quality risks, both functional and non-functional, that relate to the different user stories in the backlog. In some cases, quality risks transcend those user stories, being related to emergent system behaviours. Performance, reliability, usability and security are often emergent properties of the system. In addition, there can be some risks that do not relate to a specific user story but which are important for this iteration.

Once the risks are identified, the team categorises them and assesses their risk level in terms of likelihood and impact. We have already discussed one way to evaluate likelihood and impact, and we will look at another useful scale

and approach for assessing risk levels in a moment. After the assessment is done, the team assesses the overall distribution of risk ratings. You need to have consensus on the risk ratings, and you need to avoid a situation where everything has been rated as a high risk.

Once the risk levels are finalised, the level of risk determines the appropriate extent of testing. In other words, the risk ratings are an input to the estimation process. Once iteration planning is done and test design begins, the tester must pick the appropriate techniques for the risk item, based on the level of risk, and apply those techniques, as discussed earlier.

In sequential lifecycle models, quality risk analyses need occasional adjustment as significant new information comes to light. That is usually a lesser issue in an Agile lifecycle. However, you should be aware that changes occur, especially if, in your implementation of Agile, the business stakeholders are allowed to change the iteration backlog during an iteration.

In addition, be aware of the possibility that when you do risk analysis you might find problems in the user stories, since the user stories are an input into the quality risk analysis process. That should be considered yet another opportunity to detect and remove defects at the source, or as close to the source as possible, which is always a good thing in any lifecycle model.

[Figure 3.9](#) shows a template that you can use for quality risk analysis. While your approach to capturing and documenting quality risks in an Agile context will vary – emphasising lightweight documentation, of course – this template can give you some ideas. Let us review it.

First, note we use a five-point scale for rating probability (which is referred to synonymously as likelihood here) and a five-point scale for rating impact. This is a deliberate variation from the four-point scales used before, and is presented as an alternative.

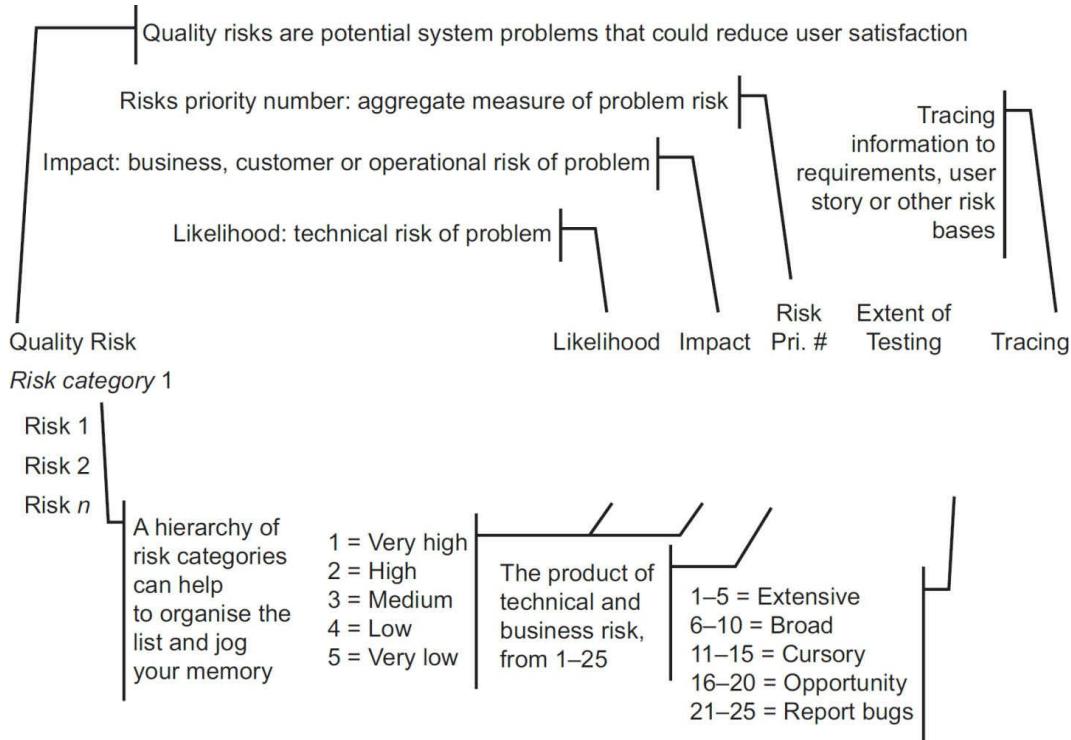
On the left-most column, you see that the template is designed to accumulate

a list of quality risks, grouped together in categories. You can use a software quality standard such as ISO 25000 to create your category list, or you can use another source of risk categories, but, either way, you should plan to customise the list to include those categories applicable to your applications. In addition, you should add examples – ideally from past bugs observed in each category – to the list. This way, when people use the categories as a checklist to identify quality risks, they have memory-jogging examples readily at hand.<sup>12</sup>

Once you have identified the risk items and assigned them into their categories, then you assess the likelihood and the impact. As shown in the template, you can use the same scale for both factors. It is a five-point scale ranging from very high to very low. For likelihood, this is straightforward. Likelihood is the probability that there is going to be a defect in the product related to a given risk item when the product is delivered for testing. Likelihood is not about how likely a user is to use a particular feature or how likely a user is to see the failure. Those are sub-factors to consider as part of impact. Likelihood is purely a technical issue.

---

**Figure 3.9 Quality risk analysis template**



You can classify something as a very high likelihood, which means it is almost certain to happen. Something that is high likelihood is more likely to happen than not to happen. Medium likelihood is something that has about even odds of happening or not happening. Low likelihood is something that is less likely to happen than not to happen. Very low likelihood refers to something that is almost certain not to happen.

Impact is rated on the same five-point scale. Impact refers to how bad the problem would be if a defect related to a given risk item escaped detection in testing and was delivered to the users. Impact ratings can be more subjective, subtle and complicated than likelihood. What is helpful is to assign specific criteria for each level of impact. Otherwise, you tend to see a real inflation problem where every risk gets rated as very high impact.

The risk priority number is calculated by multiplying the likelihood and impact ratings for each risk. Notice that each factor follows a descending scale from one to five; as the numbers go up, the risk goes down. So, with the likelihood and impact ratings shown numerically in the likelihood and impact

columns, a simple formula calculates the risk priority number.

Once you see the risk priority, you can see how the level of risk should determine the amount of testing that needs to be done and the order in which things should be tested. For a risk with a very high likelihood for bugs, when such bugs in production would have a very serious impact, then that risk needs to be tested extensively and as early as possible. With a risk that has a very low likelihood of bugs, and only bugs that few people would care about, that risk should be tested very little and very late, if at all.

In this template, you see a simple set of categories for setting the extent of testing for each risk. If the risk priority number is between one and five, you should test extensively. If the risk priority number is between 6 and 10, you should test broadly; 11 to 15 receives cursory testing; 16 to 20 gets opportunistic testing; 21 to 25 receives no specific testing, but you report bugs if you see them. This is just an example, and the ranges should be set according to the needs of the project.<sup>13</sup>

In the tracing column, you record traceability between a particular user story, requirement specification element or any other input that you use for your risk analysis. That way, if that particular item changed after the risk analysis, you can locate the affected risks that needed to be updated.<sup>14</sup>

This technique is a traditional approach to risk-based testing, adapted to Agile methods. Now, let us look at an interesting variation on quality risk analysis that combines a popular Agile estimation technique with quality risk analysis.

## Risk poker

Risk poker<sup>15</sup> is a whole-team approach to achieving optimal balance between sufficient quality and acceptable risk, on the one hand, and the available constraints in time and resources, on the other hand. We consider user stories

in the product backlog as risk items. The Scrum master organises the risk poker session, but will not participate in the game. However, the product owner will participate in the game. The product owner represents all the stakeholders and will be able to provide necessary input, most importantly about the impact. The question for each backlog item is: What level of risk should we associate with this user story?

In risk poker, we consider only two factors: impact and probability. If a user story contains more sources of risks, then the aggregated influence is considered. For example, the defect probability increases with both complexity and new technology. Instead of playing risk poker separately for all sub-factors that influence impact and probability, we try to predict the collective risk. This is our everyday behaviour: if we would like to travel into a country that is dangerous with respect to both our money and our health, we summarise these risks and decide based on the collective risk.

Some risks are not in user story level. Consider, for example, performance, security, platform or usability testing. In these cases, the testing has to be done at a different level as well. Therefore, risk poker is also played at a different level for these types of risks.

During the game, every player has four coloured cards, green, yellow, orange and red, in order of increasing risk (i.e. green is least risky and red is most risky). It is also possible to use five levels of risk by introducing another colour, though a risk from an odd number of risk levels is that people may select the middle scores as a form of mental laziness, which is not possible if the number of levels is even. The colours represent the estimated risk level for both impact and probability. Impact is estimated first, then likelihood.

After the Scrum master provides an overview of the user story, the players have the opportunity to discuss and ask questions about the story. However, people should not mention the risk level they favour, to avoid influencing each other in voting. After discussion, the team members select a card that

reflects their opinion about the potential impact of story failure, which each member places face down on the table. When every member has selected a card, the cards are revealed at the same time.

If the estimated impact risk is identical for all estimators, then that impact score is recorded. If the risk estimations are different, then the estimators with the lowest and the highest ratings explain their reasoning, followed by a re-vote.

To prevent the risk of infinite re-votes, the team can use either a time limit for risk voting on each user story, or a limit on the number of re-votes. If the limit is reached without consensus, the product owner will make the decision on the impact score.

After the risk poker has been successfully finished for impact, the players will play another game for the risk probability in the same way. The only difference is that, in the event of no consensus, the senior technical person makes the decision.

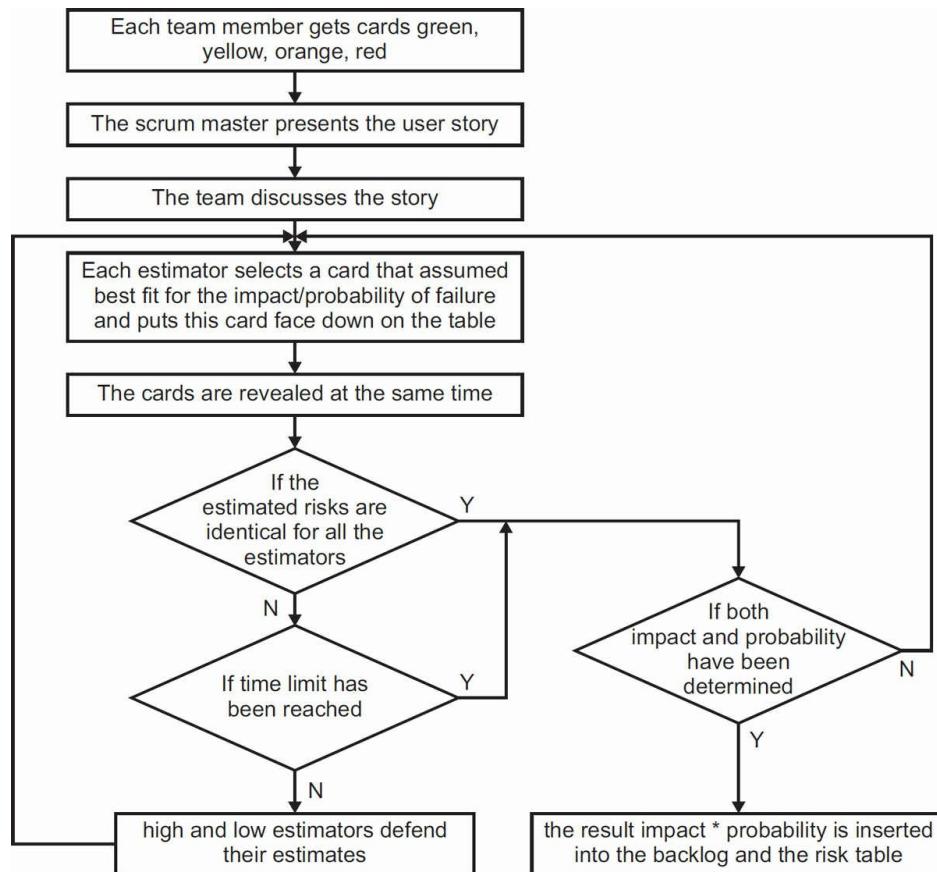
Once both scores are available, then the Scrum master records the risk points (in some methods also called the ‘risk score’ or ‘risk priority number’) into the product backlog. To do the calculations, the colours are assigned values, where the lowest risk can be the smallest number or vice versa. The risk points are calculated as the product of the two scores. This can then be followed immediately by another estimation technique, planning poker, for the same user story. We will discuss that technique momentarily.

If the risk point is too high, for example 16, then there is a high chance that we cannot optimise the costs as described earlier. In this case, the story must be broken down into simpler stories to mitigate risks and make it possible to select test techniques that will allow us to get close to this optimum.

The flow chart of risk poker is shown in [Figure 3.10](#).

As a final note on risk poker, we should point out that this approach is ritualised in the way many Agile practices are; for example, planning poker, daily stand-up meetings in Scrum and so forth. This ritualisation can help some teams adopt and adhere to best practices, but other teams see excessive ritual as overhead, or perhaps as silly. Teams that like a more lightweight approach can eschew the formality of the coloured cards and the voting for an informal discussion.

**Figure 3.10 Risk poker flowchart**



### Example 3.2.1 risk poker

After the user story has been explained, and the discussion finished, the Agile team votes for impact. The scores are:

project owner: yellow

tester: orange

developer: yellow  
business analyst: green  
Scrum master: no vote

The high and low estimators defend their scores. The Scrum master asks the tester first, who explains why she voted for orange. The business analyst explains her view as well. The next voting results in the following estimations:

project owner: yellow  
tester: orange  
developer: yellow  
business analyst: yellow

Since the time limit has been reached, the product owner makes a decision, selecting yellow in line with the majority vote.

Now, they immediately vote for probability with the following result:

project owner: orange  
tester: orange  
developer: orange  
business analyst: orange

Since the scores are identical, consensus holds, and this risk item is orange. Finally, the Scrum master inserts the risk score  $2 \times 3 = 6$  into the product backlog for this user story.

## User story risk visualisation

When all the risk estimations of user stories are available, they can be visualised so they can be seen in relation to each other. This is useful since we can see a big picture of risks. Another advantage is comparison. It is especially important to compare risks estimated in different meetings. In this way, we can easily observe a significant anomaly. For example, if we consider story 1 more risky than story 2, but in the user story risk visualisation chart we see the opposite result, it is important to validate the scores and the process itself.

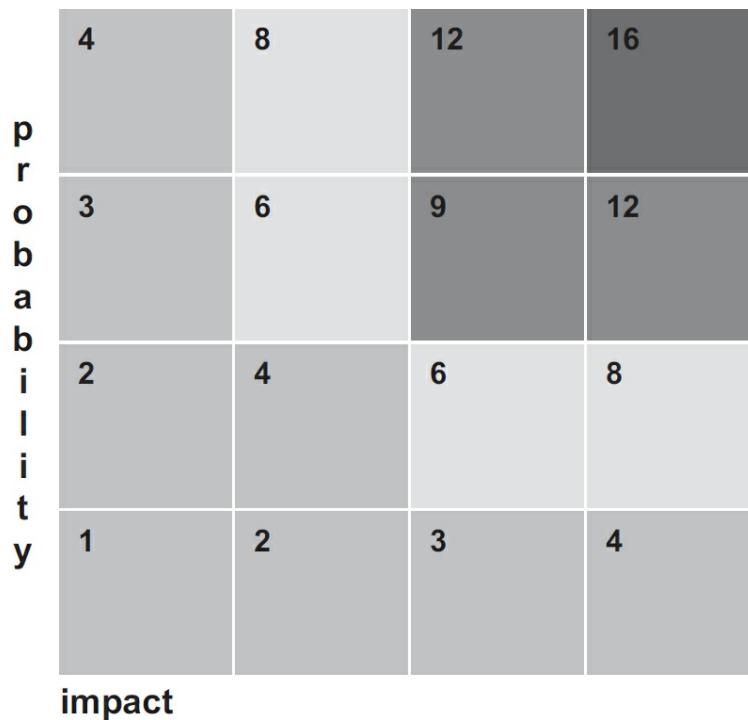
Visualisation can happen on a sheet of paper or on a spreadsheet. Following the example above, we could assign the colours of the sheet as follows: risk scores 1–4 are green; 6–8 are yellow; 9–12 are orange; and 16 is red.

However, if a company is more risk-averse, it might increase the relative orange and red area with respect to green and yellow. It is also possible to weight impact and probability differently.

The visual appearance of the story risks is a risk chart, as shown in [Figure 3.11](#). The bottom left cell contains the risk score of 1 in green (light grey in the printed version) and the upper right cell contains the maximum risk score of 16 in red (dark grey).

---

**Figure 3.11 Visualising user story risk**



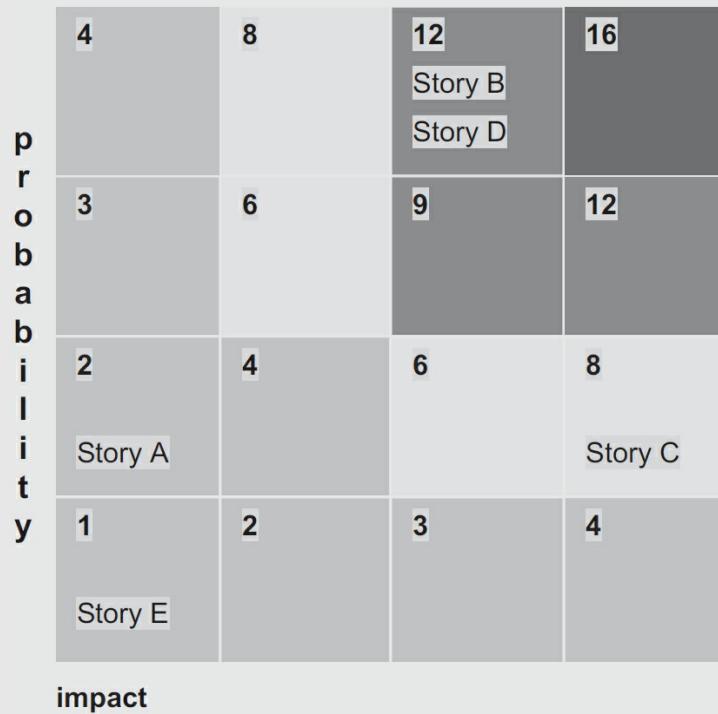
### Example 3.2.2 Demonstration of risk charts

Assume that we have five stories for which the risk pairs (impact and probability) are as follows:

1. Story A: (1, 2)
2. Story B: (3, 4)
3. Story C: (4, 2)
4. Story D: (3, 4)
5. Story E: (1, 1)

The result risk chart is shown in [Figure 3.12](#).

**Figure 3.12 An example of user story risk**



## Validating risk charts

When the risk chart has been updated, the Agile team has to validate it. One possibility is to display the distribution of the risk scores using a histogram. If this histogram is far from the normal (Gaussian) distribution, then adjustments are needed.<sup>16</sup> This technique works for quality risk analysis during release planning, but often will not work for quality risk analysis during iteration planning, since there are not enough stories.

Another method is to establish appropriate exemplars, which are user stories with agreed-upon risk ratings. These should give easily understood examples. At the very least, there should be one exemplar for each impact and likelihood rating, which would mean four or five exemplars for both, depending on the scale (1–4 or 1–5).

These exemplars allow each user story's risk ratings to be compared with agreed standards, similar to using a standardised measuring scale such as feet or metres. The exemplar stories can be selected at the beginning of the project or they can be used across projects if comparability of risk scores across projects is desirable. If the team believes that a risk prediction is wrong, then it can compare with the exemplar(s) whose impact and probability ratings are the same. If the perceived risks of the current story and the exemplar stories are significantly different, then a risk estimation error happened and the team has to reassign a better score. Obviously, one must select the exemplars very carefully as they will be used to measure risk ratings in the future.

### Testing according to risks

The main goal of the risk prediction is to test the stories according to the risk level that has been set during the risk poker or quality risk analysis. For stories with low failure probability, only a simple test is enough. On the other hand, for stories with high failure risk, a very careful test strategy is selected that contains more and different techniques. Every team's own decision is to agree which test techniques are applied for stories with different risks. [Table 3.4](#) presents one simple way to use risk to select test techniques.

---

**Table 3.4 Risk-based test method selection**

---

Story risk	Test techniques
<b>green: 1–4</b>	Exploratory test
<b>yellow: 5–8</b>	Exploratory test, defect prevention including Agile testing SBE or ATDD
<b>orange: 9–12</b>	Exploratory test, defect prevention, test design: automated or systematic based on test selection criteria, test automation, normal code coverage such as decision coverage

**red: >12** review or static analysis, strong code coverage such as modified condition/decision coverage (MC/DC) or breaking the user story into smaller ones

---

Automated test design is model-based testing (MBT), where tests are generated based on a model. Usually, the models are flow charts or state transition diagrams. The advantage of MBT is that different testing criteria can be set. However, stronger criteria may result in a proliferation of test cases. Systematic test design is equivalence partitioning, boundary value testing and so forth.

Another method defines six extents of testing: Extensive; Broad; Cursory; Opportunity; Report bugs only; None. Besides the testing techniques and the respective coverage criteria to be applied for a given extent of testing, this approach specifies the relative testing effort that should be spent addressing the risk items assigned to a given extent of testing.

### **3.2.2 Estimating testing effort based on content and risk**

Now that we have risk scores for each user story, the next step is to estimate testing effort. In Agile methods, the testing effort is estimated as part of the overall implementation effort, based on risk and the size of the user story.

Why estimate at all? In Agile, a main goal is achieving a common understanding of what the product owner wants. The team must understand the wishes of the product owner to make decisions that match those wishes. Estimation is an important part of this, since without estimates teams may plan too much work and thus fail to complete the goals of an iteration. The basic tool for Agile assessment is planning poker.

#### **Planning poker**

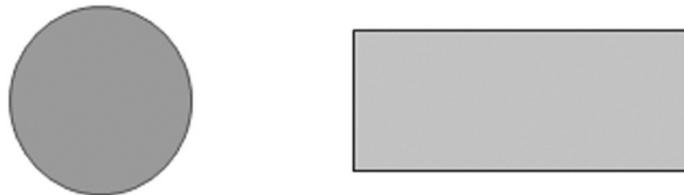
Introduced in 2002, planning poker is an Agile technique, which has become very popular (Cohn, 2005). It derives from (though some would say simply

re-brands) an estimation best practice referred to as Wideband Delphi, which was created in the 1950–1960s.<sup>17</sup> The goal (i.e. reliable estimation) of planning poker is similar to risk poker, and the processes are similar, but there are significant differences. The two methods are complementary; the planning poker follows risk poker and uses the results of the other.

Similar to risk poker, planning poker relies on relative estimation (i.e. the comparison of risks or sizes to each other). It turns out that people are much better at relative estimation than absolute estimation. For example, try to estimate the absolute area of the circle and rectangle in [Figure 3.13](#).

---

**Figure 3.13 Estimating absolute area**



Tricky, huh? However, if I ask you which is larger, you probably will respond without hesitation that the rectangular shape is. With a little squinting, you might well say that the rectangle is about one and a half to twice the area of the circle. Relative estimation is more reliable and efficient than absolute estimation.

Ultimately, we need to know the absolute prediction as well. Here we can again use comparison. If we have some historical data (and we usually do), then we only need to determine the formula to convert relative estimates to actual effort, at which point absolute estimates are available.

Instead of colours, planning poker uses numbers or sizes. One possibility is to consider a slight modification of the Fibonacci sequence, where the numbers are the following: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 and so forth.<sup>18</sup> In planning poker, we use the sequence of 1, 2, 3, 5, 8, 13, 20, 40, 100. The

reason for using the Fibonacci sequence is to reflect the uncertainty in estimating large and/or more complex stories. A high estimate usually means that the story is insufficiently understood, too complex and/or too large. Whatever the case, the story must be divided into smaller stories. Smaller stories can be more accurately estimated, and, more importantly, they can be implemented and tested more easily and better. As a general rule, if a story gets a value of 40, or even 100, then a good solution is to break it into manageable stories.

Another possibility is to use T-shirt sizes: XS, S, M, L, XL, XXL, XXXL. This metric works because we use it often and are familiar with it, unlike the Fibonacci sequence. An example is a subsequence, that is 2, 3, 5, 8, 13, 20, 40.

A key difference between risk poker and planning poker is that product owners or managers do not estimate in planning poker. This is because they might use their scores to pressure the team to overcommit. For both games, the ability to request a break, perhaps by using a special card ('I need a break') is advisable. Accurate estimation, whether of risk or effort, requires clear thought. Most people can sustain concentrated thought for about two hours, maximum.

## **Play the game**

Playing planning poker happens after risk poker is finished. We can use the result of the risk poker to estimate the testing effort, which means that pairing these two techniques together is more advantageous than applying either of them separately. In addition, since we do these techniques together, story presentation and discussion happens only once.

Planning poker starts when risk poker is complete, including all related administration. The Scrum master can play as well, but the product owner does not. Because all the necessary information is known at this time –

having just been discussed during risk poker – the team members can select a card (which they feel best fits the predicted effort of the story) and put it face down on the table. When every card is on the table, the cards are revealed at the same time.

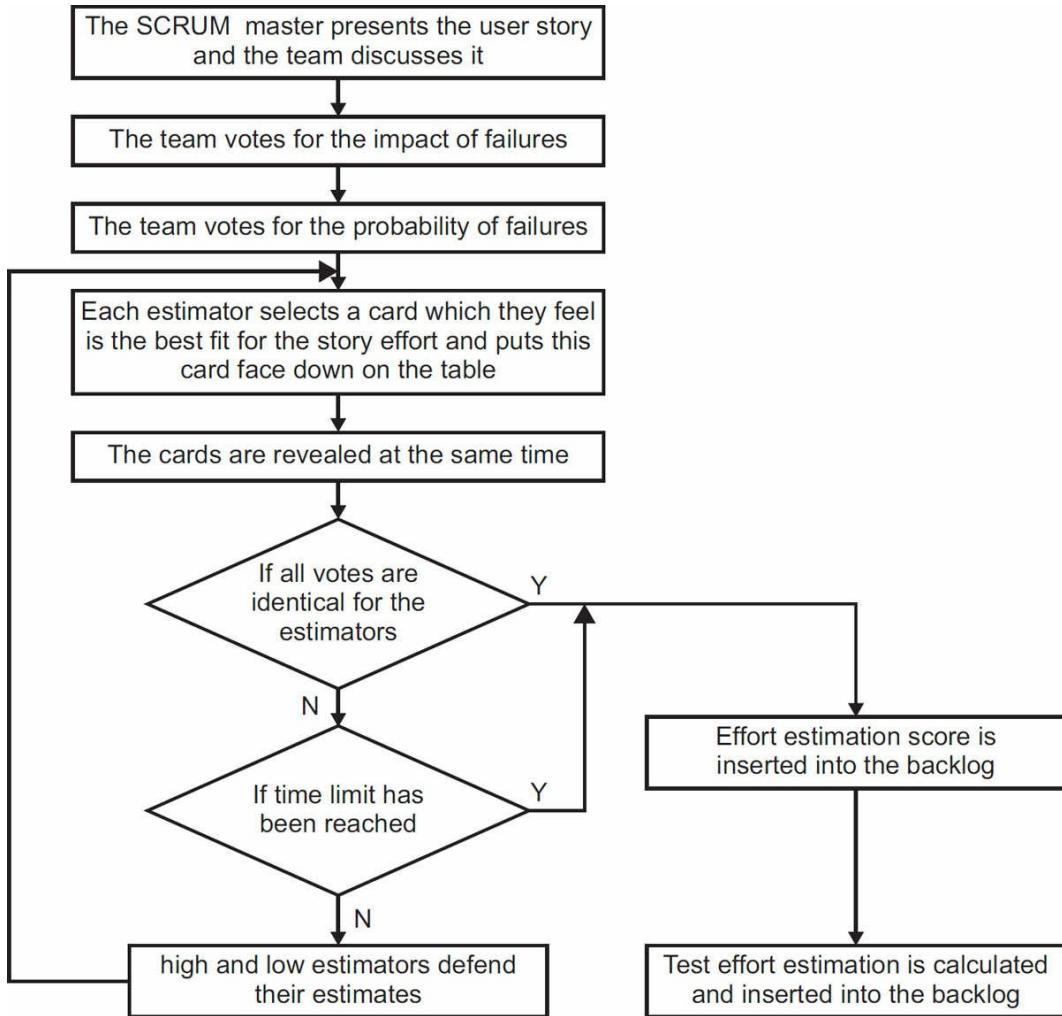
If the votes are identical for all the estimators, then the estimation has been finished and is registered in the backlog. If the effort estimations are different, then estimators with the lowest and the highest score explain their decisions.

The card selection process is repeated, until consensus has been reached, or an iteration or time limit has been reached, just as with risk poker. If a limit is reached without consensus, various approaches to breaking the deadlock exist, ranging from selecting the highest score to taking the mode, median or average score.

When the total effort has been estimated, it is time to predict the testing effort. We know that, if the risk poker results in a higher score, more extensive testing is necessary. However, with a high score in planning poker, all parts of the implementation become more difficult. The most reliable solution is to consider historical data and make the prediction based on this data and the result of both games. When everything has been done, the team will consider the next story. The whole voting process is in [Figure 3.14](#).

---

**Figure 3.14 Risk and planning poker flow**



### Example 3.2.3 Planning poker

The first vote of the planning poker gives the following scores:

estimator #1: 13

estimator #2: 40

estimator #3: 21

estimator #4: 13

After low and high estimators, that is, estimator #1 and estimator #2 respectively, have defended their estimates, the next voting round ends with the following scores:

estimator #1: 21

estimator #2: 40

estimator #3: 21

estimator #4: 21

The time limit has been reached without consensus. In this case, the highest score is selected, which is 40. This leads to the conclusion that the story is too complex. After breaking it into three separate stories, the team gives effort estimations of 8, 8 and 13.

#### **Example 3.2.4 Estimating test effort**

As described, the team broke down the complex story in [Example 3.2.3](#) into smaller stories. Since the stories are new, both risk and then planning poker must be re-executed for each story. The first new user story gets scores (orange, orange) for risk poker. Then planning poker is played, resulting in the total score 9, and effort score 13. The team has a considerable amount of historical data. It selected two stories for which the same score pair (9, 13) happened. A very useful metric is the ratio of the testing effort (cost) and the total implementation effort (cost). For the first one, the testing effort/total effort was 0.18, while for the second one, the related testing effort was 0.22. The team, therefore estimates a  $(0.18+0.22)/2 = 0.2$ , i.e. 20 per cent relative effort needed for this story.

## **Summary**

In this section, we have considered how and why risk-based testing is important in Agile projects. Risk-based testing allows Agile teams to decide how thoroughly a given user story must be tested. In addition, we saw that more thorough testing not only means more effort and more test cases, but it also means that stronger coverage criteria applies across a larger number of different testing techniques.

One way to estimate quality risks in Agile projects is risk poker. Risk poker is fun and results in acceptable outcomes if played well. Risk visualisation is important, though, as we must efficiently compare the risks of different user stories. We demonstrated the method with appropriate examples. We also showed how the risk distributions can be validated.

Finally, we considered planning poker, which is a widely used tool for estimating effort for user stories. Planning poker has to be played after risk poker. We showed an example of how to use the two pokers together.

## **Test your knowledge**

The answers to these questions can be found in the Appendix.

## **Agile Tester Sample Questions – Chapter 3 – Section 3.2**

### **Question 1 FA-3.2.1 (K3)**

Which of the following scenarios correctly illustrates the risk poker process?

#### **Answer set:**

Scenario A. The first iteration of impact failure estimation results in the following votes:

Tester: 3

Developer: 2

Business analyst: 8

After the developer and the tester defend their score, the next iteration results in a consensus

Scenario B. The first iteration of impact failure estimation results in the following votes:

Tester: red

Developer: red

Business analyst: red

Since consensus is reached, the result is stored in the product backlog

Scenario C. The first iteration of impact failure estimation results in the following votes:

Tester: red

Developer: red

Business analyst: yellow

Red has been selected, since two of the three estimators voted for that Scenario D. The first iteration of impact failure estimation results in the following votes:

Tester: red

Developer: orange

Business analyst: yellow

The Scrum master selects orange since it is the mean of the votes

### **Question 2 FA-3.2.2 (K3)**

Consider the following user story:

As the product owner, any data in the system has to be backed up and could be restored in the case of any losses.

During the second iteration of the planning poker session, the following story points were given:

Scrum master: 13

Developer: 21

Tester: 40

What is the best outcome for this planning poker session, assuming the time limit for estimating this story has been reached?

#### **Answer set:**

- A. Since the time limit has been reached, there is no possibility for another round. The Scrum master selects 13, since this is the smallest estimated value
- B. Since the difference is still too large, another iteration has to be forced. Prior to the third iteration the Scrum master and the tester

- should defend their scores as their votes are the lowest and highest scores respectively
- C. The members vote if another iteration has to proceed or not. If not, then the highest score 40 will be the accepted story point
  - D. Since the time limit has been reached there is no possibility for another round. The highest score would have to be selected; however, the score is too high. Therefore, the team breaks up the story into back-up and restore stories

### **Agile Tester Exercises – Chapter 3 – Section 3.2**

#### **Exercise 1 FA-3.2.1 (K3)**

Assume that during the second iteration of risk poker the following votes happen:

Tester: orange

Developer: orange

Project owner: red

Business analyst: red

What does the Scrum master do?

#### **Exercise 2 FA-3.2.2 (K3)**

After the first iteration of planning poker, the estimators' score is the following:

Tester: 8

Developer: 5

Scrum master: 5

Business analyst: 3

What happens next?

### 3.3 TECHNIQUES IN AGILE PROJECTS

The learning objectives for this section are:

FA-3.3.1 (K3) Interpret relevant information to support testing activities

FA-3.3.2 (K2) Explain to business stakeholders how to define testable acceptance criteria

FA-3.3.3 (K3) Given a user story, write acceptance test-driven development test cases

FA-3.3.4 (K3) For both functional and non-functional behaviour, write test cases using black-box test design techniques based on given user stories

FA-3.3.5 (K3) Perform exploratory testing to support the testing of an Agile project

This section addresses one of the core sections of the syllabus, dealing with tasks that Agile testers must carry out almost every day. Agile testers seek and interpret relevant information to create tests and evaluate test results. Especially when determining how to evaluate test results, Agile testers must work with business stakeholders to understand what correct behaviour would look like. Agile testers take user stories as a major input to writing black-box test cases and, in some teams, transforming those into ATDD tests. These tests serve to verify the software's conformance to requirements. Finally, as explained in earlier discussions about the testing quadrants, Agile testers also apply exploratory testing as part of the validation of the software.

In this section, you will notice significant overlap with the test techniques and test levels on sequential projects. However, there are some differences. The quick pace of iterations and the iterative nature of Agile creates some of the differences. Terminology, unfortunately, can also differ, which requires some mental translation. These terminological differences do not create any additional insight into testing, but are merely an unfortunate legacy of the fact that most of the people who created Agile methods were not testers.

## Determining the relevant test bases

Let us look at sources of important and relevant information for us as testers, particularly to use as inputs for our test development efforts. As specified in the Foundation syllabus, the term for these inputs is ‘test basis’, the plural of which is ‘test bases’.

The test basis, or test bases, is what we design and implement our tests against and measure our coverage against. As discussed previously (see [Sections 1.2, 2.1, 3.1, 3.2](#)), a common source of testing ideas in Agile projects is the prioritised set of user stories in the release and iteration backlogs.

However, a common mistake is made by testers around the world, regardless of lifecycle. This mistake is to have a unidimensional test basis. Having a unidimensional test basis means that you pick a single input, such as a set of user stories, and decide that, so long as this input is covered, you have adequately covered the product. That is a fallacy, whether following a sequential lifecycle or an Agile lifecycle.

Whether you are looking at a requirements specification, or a set of user stories, those items are always an imperfect reflection of what needs to be tested. Obviously, you do need to test the user stories adequately, just as you must test the requirement specifications adequately in a sequential lifecycle model. However, that is not enough by itself.

Further, in Agile methods, documentation is more lightweight. If you are used to getting detailed requirement specifications, you might at first be surprised by the limited nature of the available information. It might be that all you see are a set of user stories that have been prioritised and placed into a release backlog.

In addition, non-functional requirements might be scant or missing entirely. People talk about using user stories to capture non-functional requirements.

However, user stories are typically expressed in the form of: ‘As an x, I want to be able to do y, so that I can accomplish z.’ For example: ‘As an e-commerce customer, I want to be able to apply discount codes on checkout, so that I can save money on my purchases.’ The language of the user story is inherently tilted towards describing functional behaviour. Some companies have challenges in terms of describing non-functional behaviour using user stories. As a result, the non-functional behaviour does not always get described very well.

This absence of clarity in terms of non-functional behaviour is an important issue for testers. If you do not force this issue into the open with your developers and other stakeholders, it can result in a gap in your testing. Important attributes such as performance, reliability, security and interoperability might not be adequately tested. One way to address this problem is to make sure that your quality risk analysis process, described in the previous section, includes non-functional quality risk categories.

In addition to the risks identified through quality risk analysis, what other inputs to our test design and implementation process are necessary? It is not possible to give an exhaustive list for all projects and products, but here are some examples to get you thinking, to help you transcend the unidimensional test basis fallacy.

You should consider testing for defects that have been found on previous releases, in similar products, and so forth. Consider testing parts of the product that are known to be potential problem areas. A complete list of the system’s major functions and features, including existing functions and features that might need regression testing, should be one of your test bases.

To reduce the risk of omitting non-functional testing, your test bases should include a list of the relevant quality characteristics, perhaps using ISO 9126, ISO 25000 or some other useful source of non-functional characteristics.

Consider user profiles and user personas. For example, you might have role-based security in your system. In that case, you will want to consider whether you have tested the features that behave differently based on the role of the person using them. User personas go beyond that, though. Different people interact with software in different ways. Consider whether you have identified those different types of personas, and whether your testing covers each persona.

Increasingly, software is built to be used in different supported configurations. For example, if you are building a browser-based application, there are many different browsers, browser versions, browser security options, settings, extensions, updates and support packs. There are also anti-malware packages that can interact with the way browser-based applications work. These packages also have different settings that can interact with the way those browser-based applications function.

Consider any standards that are relevant to what you are testing. For example, if you are testing a Food and Drug Administration (FDA) regulated product, then you will need to include the FDA regulations as one of your test bases.

User documentation, marketing or marketing claims, brochures, web material, media announcements (including social media statements about the product), marketing collateral and sales collateral are additional test bases. If we are making claims about the product or system, we should test whether the software actually works as claimed.

Again, this list is not intended to be exhaustive, but rather to get you thinking about potentially relevant sources of information about what to test. In an Agile project, as in any project, do not fall for the unidimensional test basis fallacy.

## **When a user story is done**

Let us return to the DoD, which is an important concept in Agile projects. For a user story, it is done when it is ready to be passed on to the programmers to start creating the code and to the testers to start implementing the tests. So, what would need to be true about such a user story?

A user story that is done should be consistent with the other user stories in the iteration. It should also be consistent with the overall product theme. This includes any issues of release, such as releasing in a particular way or in a particular order. It also includes the look and feel, the usability aspects, of the various features. The user story should make sense in the context of the bigger story about what the product looks like.

For a user story to be ready for programming and test development, everybody in the Agile team should understand it. If anybody has questions about the user story, including the tester, it is not done. This includes having a set of acceptance criteria that are sufficiently detailed that the team, including the product owner, can clearly recognise passing behaviour and failing behaviour. As testers, we should agree that these acceptance criteria are testable; if not, the user story's not done. Any issues with the acceptance criteria, including testability issues, need to be addressed before the user story can be used as the basis for test implementation or for code implementation.

As mentioned in [Section 1.2](#), one way of thinking about a user story is in terms of card, conversation and confirmation. If a user story is done, the card should be documented adequately. We should have had a conversation between the developer, the tester and the product owner about what the story means. These three amigos should agree on how to confirm that the user story works properly.

A user story that is done at least has the basic design of the acceptance tests in place, and ideally would have the tests ready for implementation via an automated technique. We have already talked briefly about ATDD in [Section 3.1](#), and we will return to this topic in-depth in this section. Part of finalising

the user story is to design these acceptance tests.

Finally, a user story is done when we understand all of the tasks that must happen during this iteration to implement the user story. These tasks should be estimated. Together with the other user stories to be done within this iteration, that estimate should fit within our actual, achievable velocity. As mentioned in [Section 3.2](#), the estimates must include both development tasks and test tasks.

### **Example**

Suppose you are testing a browser-based application that allows people to pay their bills online. For example, these bills could be for a utility company that offers refuse pickup, electricity, water and natural gas services, each of which are separate accounts. Assume that someone might buy one, two, three or all four services, with each service being its own account. Each account is separate, can be managed separately and can be paid separately.

For this application, a business stakeholder suggests the following user story:

As a customer, I want to be able to open a pop-up window within the ‘enter payment amount’ window, which shows the last 30 transactions on my account, with backward and forward arrows that allow me to scroll through my transaction history, so that I can see my transaction history without closing the ‘enter payment amount’ window.

This feature allows customers to compare what they paid in various periods in past years.

As a tester, you would now work with the business stakeholder and the developer to create sufficiently detailed and testable acceptance criteria. At that point, you can design a set of user story acceptance tests.

### **Testable acceptance criteria**

As discussed, in order for a user story to be considered done, we need to have testable acceptance criteria in place. So, what is true of testable acceptance criteria? Let us look at some relevant attributes.

If there are externally observable functional behaviours, then the acceptance criteria should address those behaviours.

If there are relevant quality characteristics, the acceptance criteria should address those. This is especially true of non-functional quality characteristics, such as reliability, usability, maintainability, portability and efficiency. Efficiency includes performance characteristics, such as response time, throughput and resource utilisation under load. Portability is sometimes referred to as compatibility or configuration testing and includes issues such as working properly on all supported configurations.

If the user story contains a use case, or is part of a larger use case, the acceptance criteria should address the normal paths and exception paths in that use case. You want to avoid the mistake of only addressing the happy path as this usually results in users encountering strange and undesirable behaviours when the untested exception paths occur under real-world conditions.

If the user story involves interfaces, whether user interfaces, interfaces with other systems, interfaces to a database or data repository, hardware interfaces or any other interface, then the acceptance criteria should address the interfaces.

The user story might infer constraints on design or implementation, such as a particular type of interface to a particular type of database management system. In such a case, there would be a limit on how many users can be connected at once and how many applications can make queries on that system at once. Consider the negative cases, too, such as what should happen if a query fails, for example by timing out.

The user story might also imply certain issues related to data formats. For example, if dates are involved, then we need to consider the format of the date, whether day/month/year, month/day/year or however else the dates are presented. Currency information is another example. This includes the precision of the data. For example, currencies can be shown to the nearest dollar or to the nearest penny. It can also include the currency itself, such as

whether we are dealing with pounds, dollars, euros, roubles or some other currency.

In terms of documenting the acceptance criteria, you can use text if that is the easiest way to do it. However, remember that part of being Agile is minimising documentation. So, if it is quicker and easier to draw a graph, then draw a graph. A graphical depiction of equivalent partitions or boundary values could be a perfectly acceptable way to represent acceptance criteria.

## **Other sources of information for test development**

What other information might you need as a tester while you are going through the process of designing and implementing tests for a user story and its associated acceptance criteria? Let us look at some examples.

We talked about interfaces associated with the user story itself, but how about the interfaces you will use to run the test? Do not assume that is always or only a GUI. You might use a graphical user interface in conjunction with a command line interface, an application program interface, a database interface, a web service interface or a virtualised instance of such an interface. Or you might use one of these other interfaces alone or together without a graphical user interface at all.

You should also consider whether you are going to use tools as part of your testing. If so, do you have those tools? The issue of tools would have been addressed during release planning, so ideally there are no surprises here. The list of tools you need can change and evolve.

You should make sure that you know how the users – and by extension, you as the tester – will use the system. Your test basis documents – for example, the user stories – should give you information on this topic, but they might not always do so, or they might provide information that is incomplete. For example, it is not unusual for Agile testers to lack screen prototypes for new

user interfaces, use cases for new features or flowcharts or decision tables for complex business rules. If no one gives you this information, you should try to track the information down. If no one has what you need, consider building it yourself as part of the exercise of developing your tests – in collaboration with your business stakeholders and developers, of course.

You also need to make sure that the DoD for the user story is clear, including what *done* would mean with respect to sufficient testing. For some Agile testers, unfortunately, the DoD from a testing point of view can be: ‘Well, we’re out of time to test this iteration.’ This results in significant and poorly understood residual quality risk when such systems go into production. It is not clear at that point how well any given item was tested, which is inconsistent with testing best practices to say the least.

As we have mentioned throughout this book, in an Agile project you might receive less information and documentation than you are used to. You might find yourself relying on your experience, skills and knowledge more than you did when working on sequential lifecycle projects. So, you will need to consider whether you have the experience, skills and knowledge to develop and execute the tests needed for each user story. If you do not, you will need to know where to get that information from.

One possibility is to use pair testing to address the occasional skills gaps. However, if it is a bigger skills gap issue, then you will need to plan to address that more comprehensively, through training, mentoring, self-study and the like. Such skills gaps should be identified during release planning, since suddenly discovering such gaps at the start of an iteration is going to be a significant problem. The damage will be both to your credibility (always a precious asset for testers) and to the team’s ability to progress. It can take a while to learn something well enough to be able to test it. While people are supposed to be willing to role-shift on Agile teams in order to ensure progress, in the long run the team is counting on you to handle the testing

tasks. Otherwise, why have a tester on an Agile team?

While you can try to proactively avoid skills gaps, you should expect that information gaps will arise. Situations will come up where you do not have enough information to decide how something is supposed to work, even though you do have the skills and knowledge to test the application. In a sequential lifecycle model, theoretically, business analysts, requirements engineers, designers, architects and other business and technical stakeholders would deliver all the information about how something should work to you as a tester. The sequential model information flow is a push model, rather than a pull model – at least such is the theory. Many professional testers have never actually worked on a sequential lifecycle model where they received all the information they needed without having to pull it from somewhere or someone.

In Agile lifecycles, you should expect the pull model to dominate both in theory and in practice. You need to work collaboratively with business and technical stakeholders to gather the information that you need for your testing. You should expect pulling information to be an ongoing process – which hopefully is not as difficult or painful as pulling teeth.

## **Measuring done, together**

You should also expect to be actively involved in measuring the DoD for all testing activities. As in a sequential lifecycle, the tester is a source of information. In Agile lifecycles, this includes interpreting the relevant criteria in the DoD. You need to be the expert on testing and quality, evaluating test-related and quality-related criteria and the DoD.

As you evaluate test-related and quality-related criteria, understand that activity is different from being a quality cop. You are not there to enforce a particular level of quality. You are a coequal member of the team. This gives you the right to say that a particular criterion is not met, or that a particular

item does not meet its DoD. You need to be ready to explain to people why that matters.

However, avoid assuming, taking on or accepting the role of some kind of project superhero, ‘Quality Man’, with a green cape emblazoned with a bright red ‘QM’ and a stentorian way of saying: ‘This software shall not pass.’ You are part of the team. Your opinion about whether something is ready for users matters, but if you start thinking the team needs your *permission* to deliver software, you might want to check your ego to see if it has become overinflated.

Is the point of the acceptance criteria process to nail down the business stakeholder, to force them to commit to certain behaviours? In other words, do you want to ensure that the users cannot come back after something goes into production and say: ‘Hey, we didn’t expect it to work this way!’ Are the acceptance criteria a mechanism to create documentation that supports why a feature was coded and tested a certain way?

In some organisations, acceptance criteria can serve that purpose, though it is not consistent with the Agile principle of customer collaboration over contract negotiation. There is nothing magic about Agile methods. Further, the typical user does not really care much about whether the lifecycle is Agile or sequential. They simply want what they want when they want it, and when they feel they have not got it, they complain.

Assuming you have an engaged business stakeholder or product owner during the process of defining the user story and acceptance criteria, then the testers and developers are insulated to some extent from this complaint. However, sometimes the business stakeholders do not effectively or fully engage in the process. Some business stakeholders complain about Agile methods, saying that they are tired of having to hold the development team’s hands throughout the entire development process. Some of them say:

Look, we like the old sequential model, where we tell them what we want at the beginning of the project, they go off and build it, and then they give it to us. Why do we have to baby-sit these people? Aren't they professionals?

In organisations where this mindset exists among the business stakeholders, the organisation will miss opportunities to fully exploit the potential of Agile methods.

## Example

Let us continue our example of the pop-up window to view historical transactions.

To refresh your memory, here is the user story:

As a customer, I want to be able to open a pop-up window within the 'enter payment amount' window, that shows the last 30 transactions on my account, with backward and forward arrows that allow me to scroll through my transaction history, so that I can see my transaction history without closing the 'enter payment amount' window.

Here are some acceptance criteria that you could define for that user story:

- The pop-up window should initially show the 30 most recent transactions. If there are no transactions, then it should display 'No transaction history yet'.
- The pop-up window should provide backward scroll and forward scroll buttons that move 10 transactions at a time.
- Transaction data is retrieved only for the current account, not any other accounts that the customer might have.
- The window displays and populates within two seconds of pressing the 'show transaction history' control.
- The backward and forward arrows are at bottom.
- The window conforms to the corporate UI standard.
- The user can minimise or close the pop-up through standard controls at the upper right corner of the window.
- The window properly opens in all supported browsers (a reference to the list of support browsers would have to be provided here).

If you look at the user story itself, you might ask, 'I don't see how these acceptance criteria follow from the user story?' Simple answer: they don't. Remember that part of defining the acceptance

criteria is to work collaboratively with the business stakeholder and the developer. During this process, you remove (some) ambiguity from the user story by defining the acceptance criteria.

This process can lead to some interesting discussions. For example, during the writing of this chapter, one of the authors said:

Why not extend the scrolling from 10 transactions to 30? After all, originally 30 transactions fit the screen, so the next 30 will also. My own online banking systems permits 20.

There is no pre-defined right answer to this question. The right answer depends on the consensus achieved by the stakeholders. Raising and resolving such questions is part of this process. The process does not remove all ambiguity. There are further ambiguities in the user story. In a moment we will look at ways to remove these through the definition of tests.

In the meantime, let us look at each of these acceptance criteria. Some are functional elaborations of the user story:

- The window is initially populated with the 30 most recent transactions. If there are no transactions, it should display ‘no transaction history yet’.
- Pressing the backward scroll button will scroll back 10 transactions. The forward scroll button will scroll forward 10 transactions.
- Transaction data is retrieved only for the current account.
- Transaction history shows for only one account, even if the user has multiple accounts.

These are not statements found in the original user story. They arise from a conversation with the developer and business stakeholder in this process. The last two items are implied by the user story, but it is somewhat ambiguous, so this criterion makes it unambiguous.

Some of these acceptance criteria establish non-functional requirements:

- The transaction history displays in two seconds or less, which is an example of a non-functional requirement.
- The backward and forward buttons are at the bottom.
- The window conforms to the corporate user interface standard.
- The user can minimise or close the window through standard controls.
- The window opens properly in any supported browser.

As you can see, these acceptance criteria refine or extend the user story, or make clear what the story means. All of these criteria address topics that must be defined in order to develop and test the feature.

## Black-box test design

Let us move on to the topic of black-box test design techniques in Agile projects.

First, in the ideal case, in Agile projects, as well as in sequential projects, tests should be designed before the programming starts because the tests serve as a way of modelling how the system ought to behave. Black-box tests, as usage and behaviour models, help to further remove ambiguity from the user story. Following the test development process described in [Chapter 4](#) of the Foundation syllabus, in Agile projects you will use the acceptance criteria and other test basis elements as your test conditions, and elaborate those into test cases during test design.

During this process of test design, questions will arise about what exactly the user story or one of its acceptance criteria means, or how exactly something would work. Once these questions are resolved, the answers become embedded in the tests, making the tests themselves the most complete specification of the system. Notice that this is simply the testing principle of early testing and quality assurance, which applies regardless of lifecycle.

The Foundation syllabus and the Advanced Test Analyst syllabus discuss a number of black-box test design techniques. These techniques are all applicable for Agile projects. For the Agile exam, only those techniques covered in the Foundation syllabus are examinable. However, to be a fully effective tester, whatever the lifecycle, we highly recommend that you proceed to the Advanced Test Analyst level.

As an example of the application of such techniques, consider boundary value analysis. If you are looking at an integer input field, one that has clearly stated upper and lower valid values, then you will need to test the upper and lower values. You will also need to test the invalid boundary values. In addition, you should consider equivalence partitions; for example, inputting non-integer values such as decimal numbers, letters and special characters.

You can also use boundary values for non-functional test development, such as identifying the maximum number of users for performance testing. You should test at this number of users, but also beyond this number to see how system performance degrades under high levels of load.

## Example

To illustrate the use of black-box test techniques, let us return to the example of the pop-up window, and some of the acceptance criteria we have defined for that user story. Specifically, let us look at the following criteria:

- The pop-up window should initially show the 30 most recent transactions. If there are no transactions, then it should display ‘No transaction history yet’.
- The pop-up window should provide backward scroll and forward scroll buttons that move 10 transactions at a time.
- Transaction data is retrieved only for the current account, not any other accounts that the customer might have.

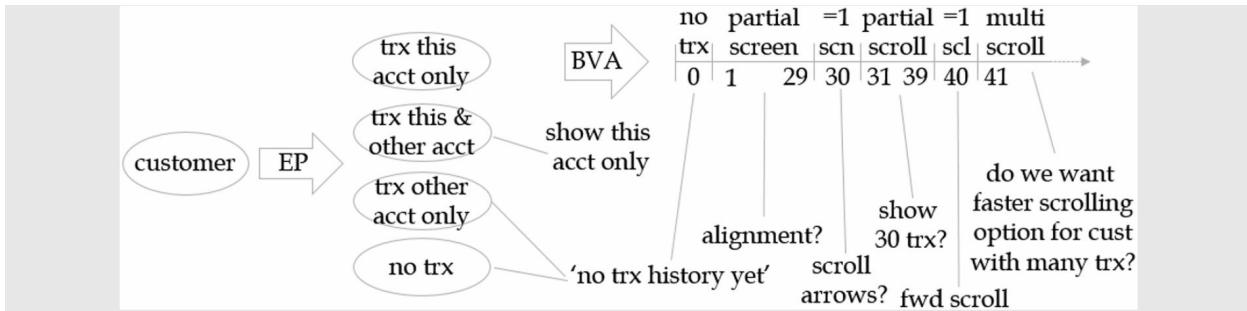
Taking those three acceptance criteria, we can use equivalence partitioning and boundary value analysis to create models for their behaviours. We can then use the systematic procedures associated with these techniques to define a set of tests.

In the course of creating the tests with these techniques, we find more questions that need clarification. You can work with the business stakeholder and the developer to capture these clarifications in the acceptance criteria or in the tests themselves. Where the clarifications occur should not matter, provided that we are collaborating with the business stakeholders and developers to answer these questions.

Let us see how this works. First, we consider the set of customers. Remember that the customers can have one or more accounts, for electricity, gas, water and refuse collection. We can partition the set of customers into four equivalence partitions, as shown in [Figure 3.15](#). One partition is for customers who have transactions only in this account. Another partition is for customers who have transactions in this account and one or more of the other accounts. Another partition is for customers with no transactions in this account but with transactions in one or more of the other accounts. Finally, we have a partition for the brand new customer, who has no transactions in any account.

---

**Figure 3.15 Example of equivalence partitioning and boundary value analysis**



**Note:** ‘EP’ means equivalence partitioning, ‘trx’ means transaction, ‘acct’ means account, ‘BVA’ means boundary value analysis, ‘cust’ means customer, ‘scn’ means screen, ‘scl’ means scroll, ‘fwd’ means forward.

For the last two equivalence partitions, the expected behaviour is straightforward. The system should display ‘No transaction history yet’. If there are transactions for this account and some other account, we need to make sure that we see only the current account’s transactions when we look at the transaction history.

We can then apply boundary value analysis to these acceptance criteria. There could be no transactions for this account. Of course, that is the same as the ‘no transactions’ equivalence partition, so that is already covered and we know what is supposed to happen.

If we have a partial screen of transactions – anywhere from 1 to 29 transactions for this account – we now have questions about the expected behaviour. Let us start with the obvious questions about display of the pop-up. What is the alignment of the transactions in the pop-up window? Do the transactions top-align, bottom-align or centre in the screen, or does the window resize automatically so that the transactions fill the screen? If the window resizes, is the resized window anchored at the top of the payment screen, anchored at the bottom of the payment screen or centred in the middle of the payment screen?

Next, how about the scrolling action? If there is exactly one screen of transactions (i.e. exactly 30 transactions) then should there be scroll arrows? For that matter, if there is a partial screen of transactions, should there be scroll arrows? If there are 31 to 39 transactions, and the user scrolls to view transactions 31 to 39, should it show exactly 30 transactions or should it show transactions 31, 32, 33, all the way up to 39? In other words, is the initial first screen a full set of 30 transactions, while the second screen is a partial screen showing only those transactions above 30?

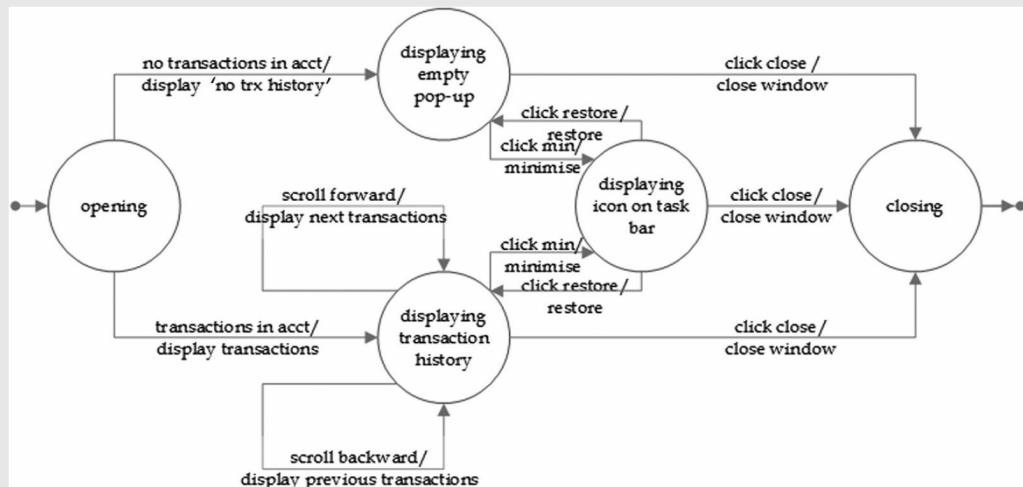
Keeping in mind that the pop-up window scrolls 10 transactions at a time, let us consider what happens if we have 40 transactions. The user can go backward and the user can go forward, of course. However, is the forward scroll button active when it does not apply? For example, if the user is seeing the first set of transactions, should the forward button be disabled, since the user cannot go any further? If the user is seeing the last set of transactions, should the back button be disabled, since the user cannot go any further?

Now, if someone has 41 or more transactions for this account, they have multiple steps of scrolling

possible. Potentially, for a customer who has been around for a long time, it could become tedious to scroll 10 transactions at a time. So, you might ask the business stakeholder whether we actually want to allow someone to scroll faster, perhaps by providing buttons that allow jumping to the beginning and to the end, or some other form of fast rewind and fast forward, as indicated on the right side of [Figure 3.15](#).

As another way of modelling behaviour – in this case the display, scrolling and screen-management actions of the pop-up – take a look at [Figure 3.16](#). If you study this diagram, you can see that it raises similar questions about the scrolling. It also raises other questions, such as whether there is a ‘maximise’ action possible. It is often useful to use multiple test design techniques as this reveals more questions – and thus opportunities to clarify the desired behaviour. Remember from [Section 3.2](#) that, in situations of higher risk, we definitely want to use multiple test design techniques.

**Figure 3.16 Example of a state transition diagram**



As you can see, test design can bring up a number of interesting questions about what the expected behaviour is under certain circumstances, even when you have a user story and a set of acceptance criteria that seem clear at first.

For those of you preparing for the ISTQB® Agile Tester exam, a word of caution. Some people, when training for the Foundation exam, do not do sufficient hands-on work with the test design techniques. Some inferior ISTQB® Foundation courses can leave you poorly equipped to move to subsequent levels of knowledge, whether at Foundation or Advanced level. Alternatively, it may simply have been a while since you applied these techniques. Whatever the reason, if you are feeling weak on these black-box

test design techniques, you should refresh your memory on the Foundation test design techniques before taking the Agile exam.<sup>19</sup>

## Documenting Agile tests

All test design techniques are applicable – white box, black box,<sup>20</sup> experience based and defect based – regardless of lifecycle model. However, the way that you document your test designs should change in Agile projects. If you are working in one of those rare organisations that, for reasons other than regulatory requirements, has chosen to follow a verbose test documentation standard such as IEEE 829 or ISO 29119, then you will be in for a rude surprise on Agile projects. There is simply not going to be enough time available to create and maintain that level of documentation.

So, on Agile projects, expect to create and use a tailored set of templates that are a lot more flexible and a lot more lightweight than those specified by these standards. Since the test documentation is more lightweight, this means that you are ultimately more reliant on the skills, knowledge and experience of the testers executing the tests.

For some people who are using Agile methods and also trying to use outsourced testing service providers, they have found the lightweight test documentation aspect of Agile creates serious challenges. These outsourced testers may have relatively limited skill and experience, at least in terms of application domain knowledge. In a sequential lifecycle, some project teams deal with these knowledge gaps by using concrete test cases for knowledge transfer, giving those concrete test cases to the outsourced test team. This approach is unlikely to work well with Agile methods because you typically will not have time to document the tests at that level of detail. Even if you do have the time to document them at that level of detail initially, you almost certainly will not have time to maintain these tests for ongoing regression purposes, unless those tests are automated using some maintainable approach

(see [Section 3.4](#)).

So, in Agile projects, beware of the documentation trap. Over-documenting is definitely a mistake that can occur in any lifecycle model, but the consequences are generally less severe than they are in Agile. The consequences are worse in Agile projects because of the ongoing change of requirements and features, which requires a tremendous amount of documentation maintenance if the tests are over-documented. What can – and often does – happen in these situations is that the overly detailed test documents become obsolete. In such a case, of course, the tests are actually less than worthless, because they incorrectly describe application behaviour.

## **Applying acceptance test-driven development**

In ATDD, we again collaborate with business and technical stakeholders to implement the tests. These ATDD tests can be evolved from the black-box test designs created earlier.

Once again, the act of developing tests will serve as a static test of the user story, as a natural outcome of analysing the user story. It will result in clarification of the user story, as specific behavioural implications are brought into focus as part of creating the tests.

In some cases, the tester will go through this process of analysis and test implementation separately, and then will bring the results back for review with the developer and the business stakeholder. Alternatively, the tester creates the tests together with the developer and business stakeholder.

The ATDD tests are examples of how the user story will work. These tests – these examples, if you prefer – should include both positive paths and negative paths, such as valid and invalid inputs. The tests should also cover non-functional elements. Remember, failing to adequately test non-functional aspects of the system is a common dysfunction – in Agile projects as well as

traditional projects – so beware of that mistake.

As mentioned, these tests are examples of system behaviour under certain circumstances. This means that the tests should be readable and understandable by all project stakeholders. This is a very critical point, and one that we must emphasise. For both Agile methods and traditional lifecycles, it is a best practice to involve developers and business stakeholders in reviewing the tests. This practice provides an opportunity for the developers and business stakeholders to see specifically what is to be tested, and how the system should behave under tested conditions. Thus, all parties go away with a clear consensus on these points.

However, this best practice can easily become a worst practice when the business stakeholders come away from the review process completely nonplussed. Sometimes, business stakeholders say,

Yes, we're involved in these test case reviews. I understand what the testers are trying to achieve with these reviews. Unfortunately, they're really not achieving it because I don't understand these tests, so I'm not able to give any intelligent comments. Further, I can't provide feedback on whether the tests are adequate, because I don't understand what I'm being shown.

Therefore, to achieve one of the main benefits of ATDD, you need to make sure the tests are written in a way that the business stakeholders and the developers will understand.

Remember how, when you define acceptance criteria, the acceptance criteria are allowed to refine and to some extent expand on the user story? That is not the case with the tests. As with the black-box test design process, the ATDD process can resolve ambiguity in the user story by providing a specific example. However, ATDD tests should not expand the user story.

## Example

Let us return to our running example for this section, the pop-up window in the ‘enter payment’ screen. We will continue with the equivalence partition and boundary value analysis tests that were designed earlier, and specifically we will focus on creating a set of ATDD tests for the navigation.

Remember, we need to test positive and negative paths, so that means stuff that is supposed to work and stuff that is not supposed to work. In order to do that, we will need some test accounts created to match the equivalence partitions and the boundary values that were shown before.

What you see in [Table 3.5](#) are the accounts that will allow us to cover the different boundary values and equivalence partitions shown previously. You also see the test cases. Each test case has a numeric identifier and a short text name. Each test case has an associated test account to be used, and it has expected results.

---

**Table 3.5 Example of an acceptance test-driven development table**

ID	Test Case	Account	Expected Results
1	No transactions	dave_smith	‘No transaction history yet’
2	Transactions other account only	linda_wong	‘No transaction history yet’
3	1 transaction this account only	jose_hernandez	Show 1 transaction, no backward scroll
4	29 transactions this account, plus other account	alisha_washington	Show 29 transactions, no backward scroll
5	30 transactions this account only	kenji_yamato	Show 30 transactions, no backward scroll
6	31 transactions this account, plus other account	hank_williams	Show 30 transactions, allow backward scroll
7	39 transactions this account	jenny_blankbaum	Show 30 transactions, allow backward scroll
8	40 transactions this account, plus other account	cosmo_banciano	Show 30 transactions, allow backward scroll
9	41 transactions this account	maxine_karolev	Show 30 transactions, allow backward scroll

There would have to be a little bit more detail to the tests, because we will have to test that we were able to scroll backward and forward and so forth. Therefore, there would be some sequence of test steps with some of the tests. For example, consider the Maxine Karolev account. For this account,

we should be able to go backward twice. The first click on the back button shows transactions 31 to 40, while the next click shows the 41st transaction. That involves assumptions about how the questions of display and navigation, raised during black-box test design, were ultimately resolved.

If one were using a Gherkin syntax<sup>21</sup> to implement the tests (such as the Gherkin syntax tests shown in [Table 3.5](#)), a portion of the table might look similar to [Table 3.6](#).

---

**Table 3.6 Gherkin format tests**

---

GIVEN 0 transaction AND customer IS ‘dave\_smith’

WHEN open Transaction window

THEN the window is empty AND no forward scroll is available

AND no backward scroll is available

GIVEN 30 transactions AND customer IS ‘kenji\_yamato’

WHEN open Transaction window

THEN the window contains all 30 transactions AND no forward scroll is available

AND no backward scroll is available

GIVEN 50 transactions AND customer IS ‘antoine\_lebeaux’

WHEN open Transaction window AND press back arrow twice

THEN the window contains transactions 41–50

AND no forward scroll is available AND backward scroll is available

GIVEN 51 transactions AND customer IS ‘sylvia\_lurie’

WHEN open Transaction window AND press back arrow three times

THEN the window contains transaction 51

AND no forward scroll is available AND backward scroll is available

---

## Designing software to pass the tests

Now, an interesting question that may have occurred to you at this point is: Well, if the tests are prepared as a team, wouldn’t the developers code to the test scripts? If so, there would be no defects, right? Well, yes, one would hope so! At least, that is the goal. Notice that what we have here is a holistic process, from user story definition through to ATDD implementation. We

have alignment between the developers, the business stakeholders and the testers about what we are building (i.e. the user story), how specifically the user story should work (those are the acceptance criteria) and what examples of correct behaviour look like (those are the ATDD test cases).

Notice that there is nothing uniquely Agile about this holistic process. It is simply an instance of the classic testing best practice, discussed in the Foundation syllabus: early testing and quality assurance. It is test design prior to code development. As always, regardless of lifecycle, early testing has the positive effect of achieving consensus with the developers and the business stakeholders about what exactly we are going to test and how the system should behave during those tests.

Of course, there are limits to the ability of this process – like any process – to achieve a zero-defect outcome. No requirements specification can be perfect, and no test can be perfect. So, we cannot say that there will be *no* defects. There is still some possibility of misunderstanding between communicating human beings, and some unavoidable ambiguity in the ultimate behaviour of the code. However, this process provides concrete, practical and efficient steps in the early days of each iteration to minimise the number of defects that are introduced during coding, discovered during test execution and resolved during the final days of the iteration.

As with any best practice, notice that we are describing how it is supposed to be done. It is not simply theory, though, as many successful organisations do follow this process. However, do all Agile organisations do it this way? Sadly, no: sometimes for practical reasons, but more often just because they are still learning to crawl before they can walk, or walk before they can run. However, this holistic process is an achievable goal to which you and your colleagues should aspire.

## **Reactive testing strategies**

We have now come to the last of the four techniques that you are expected to be able to apply in the exam – and we hope in your daily work, too – exploratory testing. Exploratory testing is one of a number of techniques, such as Elisabeth Hendrickson’s bug hunting technique (Hendrickson, 2013) and James Whitaker’s fault attack technique (Whitaker, 2002), that are associated with what the Advanced syllabi refer to as ‘reactive testing strategies’, and which are sometimes also called ‘dynamic test strategies’ or ‘heuristic test strategies’.<sup>22</sup> A reactive testing strategy is where you react to the system that is actually delivered to you, rather than trying to develop tests in advance, and you usually use heuristics as you dynamically design and execute tests at the same time.

You should always plan on some amount of reactive testing. You will never receive a complete, perfect set of test basis documents on an Agile project, or any other project for that matter. Therefore, no amount of analysis in advance will result in a complete, perfect set of tests. In addition, in Agile projects you are going to need to adapt to change. Furthermore, in Agile projects there is typically limited documentation compared to a traditional lifecycle, so you should expect a somewhat higher proportion of reactive tests, relative to pre-designed tests, than you might have in a sequential lifecycle. That proportion is raised further by the fact that you probably are – or at least you should be – doing a greater amount of automated testing, especially automated regression testing, which further reduces the amount of scripted manual testing.

When using Agile methods, a blend of four major testing strategies is generally recommended. The first two are reactive testing and analytical risk-based testing, which we have already discussed. In addition, you should include analytical requirements-based testing, which is what happens when user stories are used to develop tests, as in the last two subsections. In those subsections, we transformed a user story’s acceptance criteria into black-box tests, and then implemented those tests using ATDD. This is a classic

example of elaborating a set of test conditions into test cases, as described in the Foundation syllabus. In this case, the test conditions are the acceptance criteria, the logical test cases produced by test design were the black-box tests, and the concrete test cases produced by test implementation were in the ATDD table, [Table 3.5](#). Finally, you should also include the regression-averse test strategy, which is typically achieved through automation.

If you have forgotten where these strategies come from and what they mean, go to the Foundation syllabus. Test strategies are explained at a high level there. If you want more details, refer to [Chapter 2](#) of the Advanced Test Manager syllabus. If you really want to go in-depth on test strategies, download the Expert Test Manager syllabus, which discusses the various strategies, their strengths and weaknesses, and their application in [Chapter 1](#).<sup>23</sup>

## Example

As mentioned, you should use a blended test strategy in almost all situations, including Agile projects. Here is the blended test strategy used by one company.

The software they develop is a complex system of systems, written in Java. The developers use an automated testing tool called JUnit to produce TDD-based tests. They also use commercial static code analysis tools to evaluate the quality of the code.

Business stakeholders collaborate with the testers and developers to create feature verification tests using a tool called FitNesse.

The independent test team has two proprietary tools. They use these to create automated system regression tests. The most senior people in the independent test team are also involved in exploratory testing, looking for ways to expand their automated scripts.

Since this is an enterprise application that is customised and installed at investment banks, there are consultants involved in that installation, who are either company employees or employees of a large consulting firm that specialises in such work. They are responsible for testing that those customisations are correct. Ultimately, the banks' users perform an acceptance test.

As you can see, this company uses various types of strategies. Analytical risk-based, analytical requirements-based, regression-averse, and reactive test strategies are blended and used at appropriate points.

## **Exploratory testing**

So, returning to exploratory testing, here is an approach for incorporating exploratory testing into Agile projects. Remember that there is test analysis during the iteration planning process, which produces a set of test conditions. Some of these will be covered as part of analytical risk-based and analytical requirements-based testing, as discussed above. However, for some, the best way to test them is to include them in what are referred to as ‘test charters’. Each test charter includes one or more test conditions that the tester should cover within some fixed period, usually between 30 and 120 minutes. That fixed period is called a test session. Notice that the fixed period for the test session is another example of time boxing, a common practice in Agile projects.

So, during iteration planning, we produce a set of test charters, each with a set amount of time. The set time applies to test execution as well as time needed for the prior design and implementation work required for the test charters, such as creation of test data, acquisition of certain test props, configuration of a test environment and so forth. Such prior work must be taken into account during iteration planning.

However, in exploratory testing, the bulk of the test design and implementation work occurs once you are given the software. In exploratory testing, you design and run tests simultaneously. As your understanding of the software evolves, so do your tests. During exploratory testing, you can apply black-box, white-box and defect-based test design techniques, such as those discussed in the Foundation, Advanced Test Analyst, and Advanced Technical Test Analyst syllabi. You should augment any tests created using such formal design techniques by applying your skills and experience to imagine other interesting areas and opportunities.

Since the tests are not predetermined, there are management challenges associated with exploratory testing. You have 30 to 120 minutes to set up for

the test, to design and run the test and to do any bug investigation necessary. How do we make sure time does not get away from you in this process? A technique used to manage exploratory testing is called session-based test management.

In session-based test management, the time box for the session is estimated in advance, as discussed. During test execution, the actual time is recorded and tracked, just as you would for a pre-designed test. Tracking the time actually spent on tests, and comparing that time against the plan, allows us to apply the Agile concept of velocity to the tests.

Exploratory test sessions should go both broad and deep. Consider the test charter broadly, in terms of what it allows the user to accomplish. Next, drill down into particular aspects of the capabilities. As you do so, try abnormal and exception paths, as well as the more typical usage scenarios. Consider ways that the feature you are testing can interact with other features. Remember that you are free to cover, within the boundaries of the charter and its time box, anything that your experience and inspiration guides you to test. You have been given test conditions, but not test cases.

## **Sources of ideas for exploratory tests**

Some people find that kind of freedom overwhelming, so the test charter might give some additional hints on what to cover. For example, it can list the purpose of the charter, the actors who will use the feature and the usage personas that these users might have. To avoid false positives, the charter might give the preconditions that must be established before the test is run. Since this test charter will be run as part of a larger set of tests in the iteration, the charter may include a priority relative to other tests. The test charter can also provide references, such as test basis documents, test data to use, hints about areas to experiment with, variations of use to try and the test oracles that will determine your expected results.

In exploratory testing, because you lack the guidance of a pre-designed test, there is a risk of getting lost in irrelevant but interesting aspects of system behaviour. In order to maintain focus, you can use a number of heuristics and questions to guide your testing. First, think about the most important aspects of the system to examine during your testing. Remember that you have limited time.

Now, while finding bugs is not the only objective of testing, certainly you should look for ways to make the system fail. Try different paths through the system, especially legitimate but unusual ones. Ask yourself, as you explore these different paths, what should happen. Remember that exploratory testing is classified as quadrant three testing, which should be focused on validation. Validation is about examining behaviour from the customer and user perspective, not from the perspective of requirements. So ask yourself: Would the user or customer be happy with this behaviour?

Other under-specified, but certainly important, aspects of system behaviour are installation, upgrade, patches and uninstalls. User stories and requirements often ignore these operations, but, if they do not work it is unlikely that anything else matters.

In pre-designed testing, the test case is your guide, though you should always consider inspiration as a way to augment your pre-designed tests. In exploratory testing and other forms of reactive testing, you should rely on any skills, experience, knowledge and inspiration that you have. Some testers find that freedom liberating, and some find it intimidating. Some wonder: ‘Well, without guidance, isn’t it possible that my coverage might be insufficient?’ Yes, that is a possible problem, and it is why we recommend the use of a blended testing strategy, where analytical test strategies are mixed with other test strategies, including reactive test strategies.

During exploratory testing, there are things you can do to avoid missing something important. You should consider functional and non-functional

boundaries as you navigate through the system. You should consider different configurations, such as different supported browsers, mobile phones and so forth. You should evaluate what happens if a scenario or transaction is interrupted in the middle.

If you are using an application that works on data, you should remember CRUD, an acronym for create, read, update and delete. You should try each of those four possible operations on data, including in situations where the operation should be denied due to insufficient privilege.

## **Logging test results**

Let us return to a broader topic, the topic of logging test results. Whatever type of testing you are doing, you will need to log the results.

One element of logging – one that I consider particularly important – is the logging of test coverage. Earlier in this section we discussed the test basis documents that you should cover. Test results are not entirely comprehensible except in the context of what has and what has not been covered.

Of course, failures and other unexpected behaviours must be logged. A good rule to remember as a tester is ‘If in doubt, it’s a bug.’ Anything you find that is weird, alarming, unpredictable, irreproducible or just irritating – log it. If you are questioning whether the behaviour is correct, especially if you think a particular stakeholder would not like that behaviour, then be sure to log it.

Of course, problems that block testing, such as problems with the environments or the test data, must be logged immediately and escalated to the right parties for resolution. Anything that causes a loss of test execution time must be logged and escalated immediately, especially in Agile projects where test execution time is so limited.

In some cases, you might even need to log the actual behaviour. For example, medical devices and aircraft software is usually regulated, and testers must

log observed behaviour and compare it against the requirements. This comparison is used to save proof – auditable proof – that the system was tested against each requirement and worked as described in the requirements. This same need can extend to logging the particular tools used to run the tests.

### **More definitions of ‘done’ for Agile projects**

Let us close this section by returning to the topic of the DoD for various aspects of the project. Let us start with typical topics to address in terms of unit tests and integration tests. Both of these levels of testing are often owned by developers, so you will need to work with the developers to get them to accept these definitions of done.

For unit testing, we recommend that all code be subjected to static analysis, which should strictly check for compliance with coding standards, likely logical defects and security weaknesses. The developer should run tests that achieve at least 100 per cent decision coverage, ideally with automated unit tests that can be incorporated into the continuous integration framework. If there are any applicable non-functional characteristics, such as the time required to process a list of a certain size, those should also be covered as part of these tests. The code, the unit tests, the static analysis results and the unit test results should all be subjected to a peer review, with one of the exit criteria for that review being that no technical debt or serious defects remain in the design or the code.

For integration testing, we recommend that all unit interfaces be identified and tested. Test coverage should be risk based, and should address all functional and non-functional requirements via both positive and negative tests. Any defects found should be tracked, and all significant defects should be resolved to the extent that if the integration tests can be automated, they should be.

Moving on to the system testing level, what would it mean to have completed system testing? Certainly, end-to-end testing through various features, various functions and user stories should happen. Do not rely on feature verification testing or acceptance testing of an individual feature, because that will not necessarily test the interactions.

You should cover user personas, which refer to aspects of system behaviour such as role-based security, but it goes beyond that. Personas also refer to the ways in which people interact with the computer. Are they patient? Are they impatient? Do they tend to make many mistakes?

You should have the relevant quality characteristics defined. You should also have a definition of sufficient testing for those characteristics.

If possible, you should have tested in a production-like environment. This allows you to check for problems related to different supported hardware and software configurations. This can be a challenge for some companies because of the complexity of their environments. Now, if you use cloud computing, you can often create a new, production-replica environment at will. When production-like environments are not available, one option is to run most of the system tests in a scaled-down environment and then use risk analysis to identify those tests that need a production-like environment to work properly.

If you are doing risk-based testing, your risks should be covered to the extent that you said you were going to cover them. Your regression tests for this iteration should be automated. If the automation is done by a separate team, outside the sprint team, this will not be part of the DoD, because the DoD should only include things that could affect the end of the system testing for a particular sprint. Certainly, any defects that have been detected should be reported. Any major defects that have been found should be not only reported, but also be resolved.

For an epic or feature, which spans multiple user stories and iterations, done

implies that the constituent user stories and their acceptance criteria have all been defined and approved so we know what we were building.

Done also means that designing and coding are completed. All stakeholders should be happy with the code and there should be no technical debt. The programmers should not feel a need for refactoring of the code. Unit tests, integration tests and system tests should have been performed and done according to the agreed criteria as previously discussed.

There should not be any major open defects against the epic or feature. If there is user documentation, it should be complete. If there is any additional testing, such as security testing, reliability testing, performance testing or other non-functional testing that is needed, that should be addressed.

How about ‘done’ for a given iteration? Certainly, that means that all of the user stories and features that are part of that iteration should satisfy their DoD. We should have tested the integration of the features. We should have completed system testing for those features.

In addition, the various non-functional quality characteristics should have been adequately tested. Any user documentation should be complete, reviewed, tested and approved. At the end of each iteration, we should satisfy the system test DoD, the DoD for the features and user stories included in that iteration and the iteration criteria defined here.

If those criteria are satisfied, we should have software that is referred to as ‘potentially shippable’. This means that you could deliver the software to the customer if you wanted, and they would find that software valuable. Now, that does not mean that the software is always released when it is potentially shippable. Before the release should happen, there are other suggested criteria. For one thing, sufficient test coverage should have been achieved, both for new features and regression, to make sure that everyone agrees that the risk of release is adequately low. The development team should say they

think that the quality is acceptable, the defect discovery rate is low, the defect and user story backlog are not an impediment to release, and so forth. If you are using risk-based testing, you can evaluate and graphically display the level of residual quality risk. Of course, schedule and cost considerations can also come up here.

## **Summary**

As an Agile tester, you can use all of the test techniques and strategies that you used in traditional lifecycles. However, there are typically changes in the way they are applied. One main change is that testing happens in smaller steps as functionality is gradually implemented. This is part of the larger process of building and testing the complete product, ready for delivery.

In Agile lifecycles, testing is a critical part of maintaining an acceptable level of quality, with minimal technical debt, since we want potentially shippable software at the end of each iteration. To handle the tight timelines, Agile testers use test-first approaches. Collaboration between the tester, the developer and the business stakeholder helps to guide the development and testing of user stories throughout the iteration.

Through the use of efficient functional and non-functional test design techniques and a robust blend of test strategies, teams can create and run tests to support the evolution of the product throughout the lifecycle.

## **Test your knowledge**

The answers to these questions can be found in the Appendix.

### **Agile Tester Sample Questions – [Chapter 3](#) – [Section 3.3](#)**

#### **Question 1 FA-3.3.1 (K3)**

You are a tester on an Agile project, and you receive the following user story:

As an administrator, I want to remove an existing account's access privileges for the sales reporting component in our e-commerce system, so that we can control access to sales data.

Which of the following is a relevant, testable acceptance criterion that is specific to and consistent with this user story?

**Answer set:**

- A. The user interface must comply with company look-and-feel standards
- B. The feature must be implemented in a highly secure fashion
- C. The account should be removed from the e-commerce system after completion
- D. Custom reports defined by such accounts should remain available to other accounts after removal of privileges

**Question 2 FA-3.3.2 (K2)**

For a user story about a monthly account-activity report, a business stakeholder suggests the following acceptance criterion:

The system must generate the report quickly.

Which of the following options improves this criterion without changing the intent of the business stakeholder or the focus of the criterion itself?

**Answer set:**

- A. The system must generate the report successfully 99% of the time
- B. The system must display the first page within 0.5 seconds 95% of the time
- C. The system must complete report generation within 2.0 seconds 90% of the time

- D. The system must generate the report quickly regardless of the local language

### Question 3 FA-3.3.3 (K3)

You are working as a tester on an Agile project. You receive the following user story:

As an administrator, I want to remove an existing account's access privileges for the sales reporting component in our e-commerce system, so that we can control access to sales data.

ID	Action	Account	Expected Result
----	--------	---------	-----------------

Assume that you have an acceptance test-driven development test case table with the following columns:

Which of the following subsets of the ATDD test case table is both feasible and provides the quickest feedback on the new feature being added? Assume each row is written as '*(ID, Action, Account, Expected Result)*'.

#### Answer set:

- A. (1, remove-privilege, test-acct-1, 'Sales data privilege removed')
- B. (1, add-privilege, test-acct-1, 'Sales data privilege added'); (2, remove-privilege, test-acct-1, 'Sales data privilege removed')
- C. (1, add-account, test-acct-1, 'Account added'); (2, add-privilege, test-acct-1, 'Sales data privilege added'); (3, remove-privilege, test-acct-1, 'Sales data privilege removed')
- D. (1, add-account, test-acct-1, 'Account added'); (2, add-privilege, test-acct-1, 'Sales data privilege added'); (3, report-sales, test-acct-1, 'Sales report displayed'); (4, remove-privilege, test-acct-1, 'Sales data privilege removed')

#### **Question 4 FA-3.3.4 (K3)**

As a tester on an Agile project, you receive the following user story:

As an administrator, I want to remove an existing account's access privileges for the sales reporting system in our e-commerce system, so that we can control access to sales data.

How many tests are needed to cover the equivalence partitions for the possible variations of accounts to be removed?

**Answer set:**

- A. 1
- B. 3
- C. 6
- D. 9

#### **Question 5 FA-3.3.5 (K3)**

You are defining an exploratory test charter for a user story about a monthly account-activity report. Consider the following elements:

- I. test conditions
- II. test steps
- III. priority
- IV. test data files
- V. test basis
- VI. test oracle
- VII. expected results

Which of the following statements is true?

**Answer set:**

- A. All seven of these elements may be defined in the charter, as they provide useful guidance to the tester
- B. I, IV and VI may be defined in the charter; II, III, V and VII should not be defined in the charter
- C. I, III, IV, V and VI may be defined in the charter; II and VII should not be defined in the charter
- D. None of these elements should be defined in the charter as they unnecessarily inhibit tester creativity

### **Agile Tester Exercises – Chapter 3 – Section 3.3**

#### Scenario: Our Software Really Gets Around Town

You are a tester on an Agile team. The team is maintaining and developing an existing mobile app for multiple phone and tablet platforms. This app allows people to find nearby restaurants, pubs, bars and similar establishments. The primary target users are business travellers.

Current features include accounts with various levels of permission and implanted capabilities. All accounts, including guest accounts, can find locations based on open-ended search criteria, and there are also filters available for price, distance and ratings. If the user registers their account and verifies their identity, the user can also rate and review places they have visited. Administrative accounts not only have all these capabilities, but can also edit and delete reviews, send direct messages to registered accounts and delete accounts and block users who engage in unacceptable behaviour.

#### **Exercise 1 FA-3.3.1 (K3)**

Consider the following user story for this app:

As a registered user, I want to create lists of favourite restaurants by city,

so that I can return to them on subsequent visits.

Create a set of testable acceptance criteria for this user story.

### **Exercise 2 FA-3.3.4 (K3)**

For one of the acceptance criteria you created for the previous exercise, use one or more black-box test design techniques to design tests.

### **Exercise 3 FA-3.3.3 (K3)**

Using the tests you have just designed, or some other set of tests for this user story, create acceptance test-driven development test cases in a table.

### **Exercise 4 FA-3.3.5 (K3)**

Using one or more criteria not already covered by your black-box and ATDD tests, create an exploratory test charter for this user story.

## **3.4 TOOLS IN AGILE PROJECTS**

The learning objectives for this section are:

FA-3.4.1 (K1) Recall different tools available to testers according to their purpose and to activities in Agile projects

Appropriate use of testing tools is essential for the success of Agile projects. Projects that deliver often, change often and, on top of that, require high quality products that can be accepted by the customer are very demanding. Testers usually do not have enough time to test the whole product manually, especially in terms of regression tests, so test execution tools and continuous

integration tools are very important to testers. Some approaches to development (for example, TDD) also require tools if they are to be implemented effectively.

Basic testing tools are described in the Foundation Level syllabus, Chapter 6.<sup>24</sup> Most of these tools can be used in Agile projects. Some of them are used in the same way (bug tracking tools, for example) while others are used in different ways (such as test management tools). Some Agile teams use tools designed especially for Agile methodologies, which are designed to handle Agile specific tasks, processes and work products; for example, user stories, daily stand-up meetings and task boards respectively. Tools used by Agile testers include configuration management tools, which are used the same way as in traditional methodologies but become more important because of continuous integration and extended test automation.

We will use an example project in this section to show tools used by Agile teams. You can assume that the project is managed using an Agile methodology with two-week iterations.

### Example – gym chain scenario

In this section, we will use an example of a gym chain with about 100 gyms throughout Europe. The systems that comprise the organisation's system architecture are shown in [Figure 3.17](#).

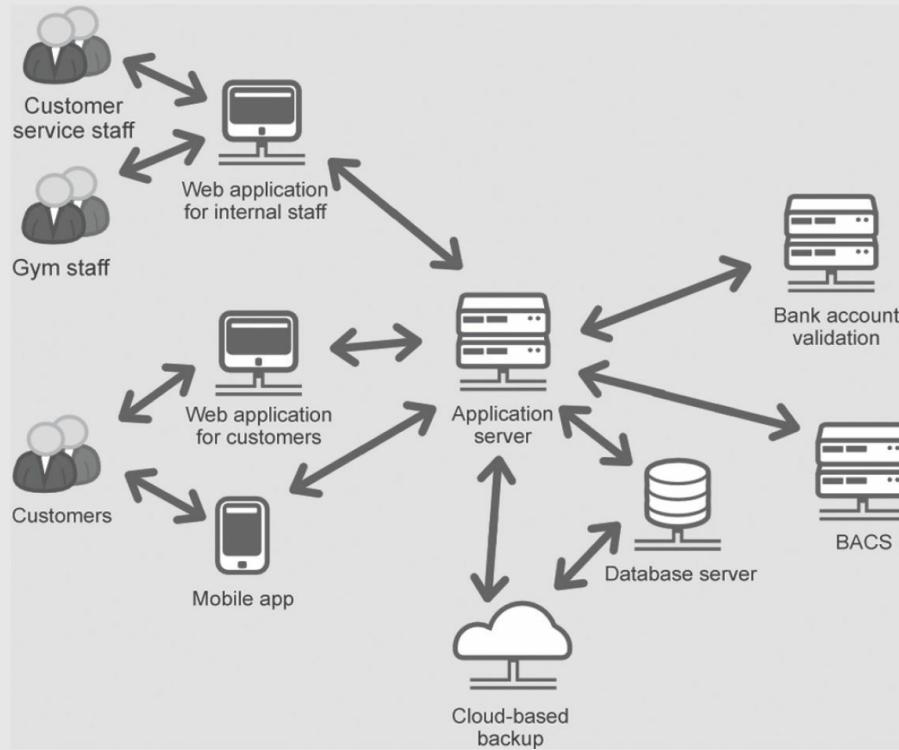
The general public can book gym hours at any of the chain's gyms by:

- Contacting staff in the gym, who then use a web application to make the booking.
- Telephoning customer service agents, who then use the same application to make the booking.
- Using the company's website to make the booking online.
- Using downloaded mobile apps from the Android and iOS app stores.

In all cases, communication with the application server is done via representational state transfer (REST) services. Direct debit payments are made via bankers automated clearing services (BACS). Files are transmitted. Confirmation and error messages are received. Credit card payments can be made. An enhancement to the security systems is being made to comply with payment card industry standards. Payments can also be made through leading electronic payment systems (such

as PayPal). Validation of bank account details is performed by sending XML messages to and from a third-party system.

**Figure 3.17 Gym system architecture**



Each sprint will deliver an ever-increasing set of functionality and features (i.e. the product increment), including the following:

- Various changes to website screens to improve usability.
- The introduction of a new third-party calendar object from which the user can select dates and times.
- Removal of the ability for customers to pay by cheque.
- An amended customer bill plus two other amended documents.
- Two new output documents.
- Fixes to various existing low- and medium-severity defects.
- Improvements to disaster recovery by using cloud-based methods.
- Enhancements to the mobile apps.

### 3.4.1 Task management and tracking tools

During iteration planning, the Agile team decides which elements (i.e. features, user stories) from the project backlog should be realised in the next iteration. They select the relevant project backlog elements for the iteration backlog. The team then plans additional tasks that need to be done while developing each feature. Some of these tasks relate to testing and other quality-related activities.

To track completion of the tasks, Agile teams usually use paper or electronic task boards. Task boards may be physical, using white boards or corkboards. In such cases, tasks are written down on index cards or sticky notes, then moved across the task board by hand. Teams may also use electronic task boards, which are sometimes displayed on large monitors in the team room.

If the team is following the rules of Kanban (perhaps as part of a blended methodology such as Scrumban), tasks move according to a ‘pull’ principle. In other words, tasks are ‘pulled’ to the next column (or station in Kanban terms) by the person who will work on the task next. The pull principle says that tasks are not ‘pushed’ into the station by a manager or another team member. Further, the Agile principle of self-directing teams says that tasks are not assigned by a manager to the person who will work on the task next, but rather that the team determines who will work on the task. In practice, a number of organisations following Agile lifecycles have chosen to ignore one or both of these theoretical principles.

### **Example**

In the gym chain scenario, the team may choose to use an electronic task board, as part of the work will be done in India. The iteration backlog and task board after planning sessions for one of the iterations may look as shown in [Figures 3.18](#) and [3.19](#). The tool being used here is Redmine with a Kanban plugin.

---

**Figure 3.18 An example of iteration backlog**

<b>0.2</b>	2016-05-16	2016-05-30	16
2038	Usability improvements	New	5.0
2040	Bugfixing Previous	New	2.0
2039	Amend customer bill	New	3.0
2047	Bugfixing previous	New	3.0

**Figure 3.19 An example of a task board**



Task management and tracking tools are used to record requirements for the product, usually in the form of a product backlog, i.e. a sorted queue of user stories that are yet to be implemented. This queue is sorted according to story priority, with more important user stories ahead of less important user stories in the queue. This use of a backlog ensures that during iteration planning nothing will be lost, and the sorting of the backlog ensures that important stories are worked on first.

These tools also track estimated story points for each user story and task. This allows the tools to calculate the total effort required to finish an iteration

or a feature. Visualisation of estimates in one place helps in conducting planning sessions efficiently. For example, if the total effort exceeds the team's average velocity (i.e. the amount of work that the team can complete in one iteration), the team knows that they must stop adding new features to the iteration backlog or risk working overtime during the iteration.

Task boards can connect tasks to user stories and features. Both development and testing tasks should be put on a task board to show all work that is needed to implement the feature. In this way, one table shows the status of all the tasks, and all team members can see how the team is doing. The task board can also show sprint impediments, i.e. tasks that block other tasks and should be finished as soon as possible.

Every team member updates their task status as soon as they start or finish work on it, to enable other team members to pull tasks through the process without undue delay. Again, if the team is working according to the pull principle, team members do not assign tasks to other members, but rather denote the status of their work on their own tasks.

Task management tools should also provide visual representation of the current status of a project. They can present charts with metrics or a dashboard, and help all stakeholders know how the project is doing. If project stakeholders are distributed geographically, teams should use software task management tools to allow easy access to task data.

These tools can be integrated with configuration management tools to allow automated recording of data; for example, code check-ins, builds of new versions and, in some cases, automated updates of task statuses.

## Example tools

The following tools can help Agile teams in task management and tracking:

- Atlassian Jira;

- Redmine + Kanban module;
- Microsoft Team Foundation Server;
- CA Agile Central;
- VersionOne;
- Trello.

### **3.4.2 Communication and information sharing tools**

The Agile Manifesto states that project teams should value individuals and interactions over processes and tools, so Agile project team members should communicate mainly face to face. Daily stand-ups serve as the main place for updating status and raising problems. Such communication allows Agile teams to avoid building excessive documentation.

Nevertheless, some information should be written down because not all knowledge can be held in peoples' heads without loss or corruption. Wikis are a good tool to enable the capture and exchange of important project information. They are easily edited and formatted. They also allow information to be linked together.

#### **Example**

In the gym chain scenario, the testers describe the test environment configuration in a project wiki. It could be especially useful as the mainframe team is located in India. They could change that configuration during non-working hours of other teams. This way all team members have access to up-to-date configuration data.

Testers also put test data into the wiki. For example, they store test credit card numbers, bank accounts and PayPal accounts, which they can use to test the BACS.

Project teams can use wikis to store information about tools and techniques. They can store templates of documents and reports as well as procedures (for example, deployment or installation procedures).

Wikis allow the easy linking of items together. Teams can leave placeholders

for information to be gathered in the future. Wikis can also store files and pictures. This makes wikis good places to store product feature diagrams, feature discussions, prototype diagrams and photos of whiteboard discussions.

Wikis can be divided between several project topics. For example, the main wiki page can contain the table of contents, which can direct the reader to design documentation, test documentation and management documentation. The latter can include metrics, charts and dashboards on product status, which is especially useful when the wiki is integrated with other tools. Wikis can be integrated with continuous integration tools and task management tools. When the wiki is integrated in this way, the status of tasks, defects and builds can be updated automatically. This way all team members can access accurate and current information about the iteration and the project as a whole.

Wikis can be used to store minutes from meetings, to-do lists, checklists, and recordings from team chats. They also allow commenting on wiki entries and tracking of discussion about those entries. Wikis are very useful tools for Agile teams, because they provide a lightweight way to note and edit information for project teams that prefer informal communication.

## Example

In the gym chain scenario, the teams (including the geographically distributed teams) can hold daily stand-up meetings through a videoconferencing system. They should agree on the most convenient time for a meeting, which can be a challenge because of time zone differences.

According to Scrum practices, during the daily stand-up meetings, every team member answers three questions:

1. What have I accomplished since my last working day?
2. What I plan to do today?
3. What is blocking me?

The answer can look like:

1. Task #2042.
2. Task #2046.
3. Nothing.

In this example, the numbers given are task numbers from their task management tool. Alternatively, the team members might use the task name.

Electronic means of communication can help geographically distributed teams. When organising face-to-face meetings is too costly or not possible for some other reason, Agile teams may use audio or videoconferencing and instant messaging. These tools allow real-time and asynchronous communication respectively. By capturing the conferences and messages, they also can keep a history of conversations. Some of these tools enable team members to share files, pictures and screens and to make presentations.

When different stakeholders or team members work in different geographical locations, desktop sharing and videoconferencing tools become very useful. Agile teams may conduct product demonstrations via a videoconference. In addition, these tools can enable code reviews, design reviews and any other type of co-working (even pair programming). Communication through such electronic channels can be captured and stored for future use or reference.

Teams should carefully choose their means of communication. Face-to-face meetings convey both verbal and non-verbal information, while emails cannot convey non-verbal content. The appropriate communication channel balances costs and needs. Electronic communication should complement, rather than be a substitute for, face-to-face dialogue.

## **Example tools**

The following tools can help Agile teams in communication and information sharing:

- Slack;
- Skype;
- Lync;

- Webex;
- GoToMeeting;
- HipChat.

### 3.4.3 Software build and distribution tools

The risks and benefits of software build and distribution tools (such as continuous integration tools) were discussed in [Section 1.2](#). You should refer to that section for more information. Here we are discussing the implications of these tools.

#### Example

In the gym chain scenario, the project team prepares automated regression tests for the REST services and adds them to the daily build process. Any change in the middleware is tested and any failure in implementation is immediately discovered. That reduces the risk of undetected integration defects in the products. Such defects can make system and acceptance tests harder, because they are more difficult to analyse and reproduce at the system level than at the integration level.

#### Example tools

The following tools can help Agile teams in software building and deployment:

- Jenkins/Hudson;
- Microsoft Team Foundation Server;
- TeamCity;
- Bamboo;
- CruiseControl.

### 3.4.4 Configuration management tools

Any software team needs to coordinate work products, identify the work

products as they are developed and store and control their versions and changes. Usually this means using configuration management tools for source code and other work products. Configuration management can cover test work products such as automated test scripts, test cases, test data and test environment configuration information.

Keeping all project work products in one repository enables the project team to trace relationships between them. A common location for all work products allows recreation of the whole project configuration for each version of a work product (for example, requirements or source code) on a given date. It is especially important when several concurrent versions exist (such as development, test, production and, perhaps, different versions for different customers).

Keeping all project assets in one repository is not always possible. When the team uses task management tools and test management tools, some work products (user stories or test cases, for example) may be stored in those tools' databases, making it hard to put those products under version control. In those cases, each tool tracks versions of its managed items. Correlation and traceability between different types of project work products must be kept manually, perhaps using a wiki entry.

Configuration management tools can scale up. Some of them require a repository on a single server and others permit distribution of repositories among several servers. Such distribution may be important for distributed teams. The team should choose the right solution based on team size, structure, location and requirements to integrate with other tools.

### **Example**

In the gym chain scenario, all the automated tests (such as the regression tests mentioned in the previous example) are kept in the same repository as the source code. By doing so, the team can correlate versions of source code and versions of automated tests. They can recreate and analyse failures in the appropriate test environments, based on their versions, if a production failure occurs.

With the geographical distribution of the teams, a version control system enables sharing of work products between teams without the need for synchronous communication. When starting new tasks, each member of a team can get the results of other tasks and build upon them without repeating the work. Each member of a team can commit their work for the use of others without the hassle of informing the whole team.

Another use of configuration management tools is allowing continuous integration tools to run build jobs only when changes have been made to source code or other work products.

## **Example tools**

The following tools can help Agile teams in versioning of code, tests and other artefacts:

- Subversion;
- Git;
- CVS;
- Microsoft Visual SourceSafe;
- Mercurial.

### **3.4.5 Test design, implementation and execution tools**

Agile projects change rapidly. Sometimes there is no time to manually perform certain tasks. In addition to rapid changes, informality within these projects prevents teams from producing extensive documentation, so test tools may prove necessary. Many of these tools are known from traditional methodologies or from outside the testing domain, but they have features that enable teams to work faster, more accurately and with less overhead.

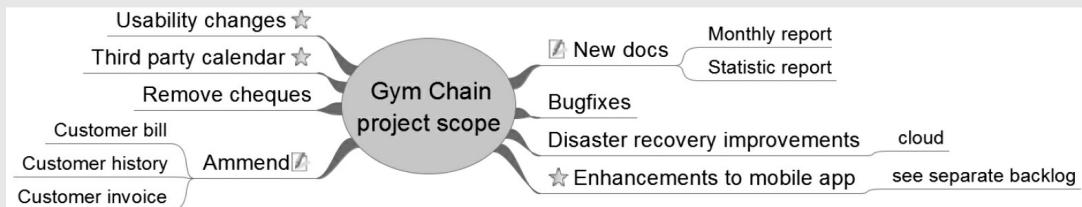
#### **Test design tools**

In addition to classical test design techniques and tools, mind maps have become popular with some Agile testers, perhaps in part because of their prominent place in some books on Agile testing. Some people find that mind maps allow easy visualisation of features or user stories, thus aiding test design.

## Example

Figure 3.20 shows a mind map drawn by a team discussing the scope of the project in the gym chain scenario.

**Figure 3.20 Example of a mind map**



## Example tools

The following tools help in drawing mind maps:

- FreeMind;
- XMind;
- MindManager;
- MindMeister.

MBT tools allow automation of test design. Here we consider only MBT tools, not the overall process of model-based testing (Kramer and Legeard, 2016). The key part of these tools is a model containing all the necessary information for the tool to generate test cases. Each tool contains the following parts:

- model editor;
- model checker;
- test case generator;
- test output generator.

A few tools contain a test execution engine as well.

The models can be created and modified with a model editor. Models can be graphical, textual or mixed. Commonly used graphical models are activity diagrams and state diagrams. However, advanced tools permit input/output values, program code and other textual elements since, originally, these diagrams did not use test design elements in the models. Textual models are tailored programming languages, formal modelling languages, such as object constraint languages (OCLs), or test specification languages, such as Testing and Test Control Notation Version 3 (TTCN-3). Another textual model supporting Agile uses the extended Gherkin language (i.e. the ‘given x, when y, then z’ syntax used for BDD), which makes models and tests understandable for everybody.

The model checker syntactically checks the model. If the model is ready, then the test cases are generated. Most of the tools allow the use of appropriate test selection criterion based on the nodes and the edges of the graphical model. Finally, the output generator exports the tests in an appropriate format. One format is a natural language version for manual tests. Other formats are readable for different test execution tools.

In Agile testing, owing to the principle of test-first development, it is important to create the tests before the implementation of the relevant user stories. Therefore, test cases contain no implementation-related elements. MBT tools are a good fit for test-first methods. Models can be created for any user story at the beginning of each iteration. Models are easier to maintain; therefore, the derived tests are available quickly and in a maintainable fashion.

## **Example tools**

The following tools generate test cases from models:

- Conformiq;
- Agile Designer;

- MaTeLo;
- Spec Explorer;
- Tosca;
- 4Test.

## **Test case management tools**

Agile teams should use also test case management tools. They can be part of the task management tool used by a project team. If so, the tests are linked with testing tasks (such as regression) and with requirements, which allows traceability. Agile teams may also use traditional test management tools or application lifecycle management tools.

### **Example**

In the gym chain scenario, the project team adds acceptance test cases to the DoD for user stories or features. These tests can be stored in application lifecycle management tools next to user stories. When all acceptance tests pass, the user story is marked as done.

This approach helps to maintain the high quality of the software product because explicitly adding tests to the DoD will promote thinking about quality and will make requirements clearer.

### **Example tools**

The following tools can help Agile teams to manage test cases:

- TestLink;
- HP ALM;
- Microsoft Team Foundation Server;
- IBM Test Manager;
- TestRail;
- Borland Silk;
- Jira + Zephyr;
- and many others.

## **Test data preparation and generation tools**

Many projects need large sets of data, which are realistic replicas of production data (at least in ways that matter for testing) but are artificially created. Testers can use test data generation tools to prepare these sets of data. Such tools can produce pseudorandom data that covers all necessary variations for testing. These tools can also change data structures and help to refactor data generation scripts as requirements for the project change.

In some cases (such as for telecom bills, bank account history), it is hard to prepare test data from scratch. In these situations, data preparation tools can take data from the production system and anonymise them. This process produces data that is realistic and valid for testing, but does not reveal sensitive information. Using production data in testing may be forbidden in some countries. Test data preparation tools can also compare or validate large databases or other sets of data.

## **Test data load tools**

After data has been generated or anonymised, it must be loaded to the test environment. Manual entry of large data sets is time-consuming, tedious, error-prone and usually unnecessary. Data can be bulk loaded into test databases. Sometimes it requires tools to accommodate the data schema to the new database, but many data generation tools have components performing data integration.

Sometimes, performance test tools are used as data preparation and load tools. Automated scripts enter the data through the user interface, allowing the business logic of the application to take care of internal data integrity. This process is advisable only when the risk of functional failures is low enough.

### **Example**

In the gym chain scenario, testers get and anonymise production data regarding hotel customers, their visits, invoices and payments. This data is then used during unit testing, system testing, load testing and product customer demos.

## Example tools

The following tools can help Agile teams with test data preparation and loading into test environments:

- CA Grid-Tools;
- Microsoft SQLServer Integration Services;
- IBM, Oracle and other database tools;
- ARX – data anonymisation tool.

## Automated test execution tools

Because regression testing is a major factor in Agile projects, many test automation tools are used by Agile teams. All test execution tools used in traditional methodologies can be used in Agile as well. If we treat test automation as a mere change in the way tests are executed, the project management methodology does not influence the use and choice of test execution tools.

However, there are some test execution tools that are more suited to Agile testing. They enable Agile approaches to testing such as TDD, ATDD or BDD. These tools allow the creation of tests from user stories, and in some cases the expression of tests in domain or natural language (as in keyword-driven testing) or semi-natural language such as Gherkin.

TDD is used mainly by developers. This approach typically involves a unit test framework such as JUnit (for Java). These frameworks are generally called xUnit frameworks, where x stands for any programming language. There are dozens of xUnit frameworks: JUnit for Java, CppUnit for C++, PyUnit for Python, and so forth. The framework provides the environment

needed to execute a small piece of code (typically a single function, method or class) even though it is not actually executable properly alone. The framework provides drivers that are used to call the code of the module and also any stubs needed to return information to the module. The framework includes a test runner so test cases are run by the tool. As the test runs, the framework checks for and captures failing assertions. These frameworks can run unit tests against the code that does not even exist yet, to support the TDD approach (e.g. see Beck, 2002).

## Example

In the gym chain scenario, the team can use automated execution tools in many ways. They can use tools to test integration with the middleware. They can use tools to test amended or new customer documents. They can use tools to automate mobile application tests.

Test execution tools provide only limited support for usability testing, but some static analysis tools can verify some elements of usability.

## Example

Here is an example of the specification and test case written in Concordion with Markdown notation:

```
### logging in example

As a regular customer [John.Smith](- "#username") of GymChain
with password [Abcd1234](- "#password")
I want to [login](- "login(#username, #password)") to webapp
and [see](- "goToHistory()") my [previous appointments] (- "?"
=checkAppointments(#username)")
```

The above text displayed as specification would appear as follows:

Logging in example:

As a regular customer John.Smith of a GymChain with password Abcd1234

I want to login to webapp

and see my previous appointments

The Concordion tool would interpret this text as an automation script, first setting #username and

```
#password variables, then calling login and goToHistory functions, and finally checking the checkAppointments assertion.
```

Keeping the specification and tests in one place ensures that changing specifications will break the tests and force an update of the tests. This way all automated tests are always consistent with the specification.

## Example tools

The following tools are automated test execution tools:

- Behaviour Driven Development;
- Cucumber;
- JBehave;
- Concordion;
- RobotFramework;
- FitNesse.

TDD – xUnit testing tools:

- Junit;
- CppUnit;
- PyUnit;
- NUnit;
- TestNG.

Other general purpose test execution tools:

- Selenium;
- AutoIt;
- Appium;
- HP Unified Functional Testing;
- IBM Functional Tester;
- TestComplete;
- and many, many others.

## Exploratory test tools

Fast changes, short iterations and an increased degree of test automation in Agile projects result in the greater use of exploratory tests in comparison with traditional methodologies. Exploratory tests require minimal documentation, which is both their strength and their weakness. They are lightweight and easy to conduct, but hard to manage. They are very effective when the test team knows the project domain well, but do not easily provide traceability or coverage.

One of the remedies for poor manageability of exploratory tests is session-based test management. There are tools that help track exploratory test charters and logging of exploratory test sessions. They allow, for example, session logs to be combined and put into one report, which helps in assessing test coverage.

Also useful during exploratory testing are tools that capture the activities of a tester. Some record a video of the last couple of minutes of testing. Exploratory testing is very dynamic, so the tester sometimes cannot repeat all the actions and data entered to recreate a failure. Recording of the test allows the tester to reproduce an incident and check if it is worth reporting. In addition, testers can attach recordings to defect reports to help analyse the root cause of the failure.

## Example

Exploratory tests are especially useful when testing a part of a product that has many degrees of freedom. In the gym chain scenario, they are appropriate for testing amended and new customer documents. However, exploratory testing is not helpful in testing middleware integration, because middleware usually takes the form of services that implement specific algorithms, and such software components do not provide much room for exploration, but are best tested with black-box or white-box techniques.

## Example tools

The following tools can help Agile teams in exploratory testing:

- RapidReporter;
- qTest eXplorer.

## Continuous testing tools

Continuous testing tools make it possible to detect bugs as soon as possible (Saff and Ernst, 2004). Due to continuous integration, new bugs are detected once a day or quite a few times a day. Continuous testing tools can detect new bugs immediately after any code modification. The continuous testing method complements the accepted advantages of continuous integration.

The first continuous testing tools executed all the test cases after each code modification had been committed to the code repository. Since there is not enough time for executing all the tests, an appropriate execution order must be set. Newer tools, however, select only those test cases that are influenced by a modification. In this way, the more often a developer saves code, the fewer test cases are selected and executed. The most advanced tools select a test only if the code along its execution trace has been modified during the last save of the source code.

Continuous testing tools only select the tests to be executed. The tests are executed by a test execution tool or framework such as JUnit. These tools could be used for manual testing as well, if the tests to be re-executed are aggregated during a given period.

### Example

In the gym chain scenario, a developer modifies and saves a code segment. Test cases T4, T11, T53 and T92 covers this modified code. The team is using a continuous testing tool. This tool selects T4, T11, T53 and T92. JUnit executes those four tests immediately. T4, T53 and T92 pass, but T11 fails. The developer promptly recognises that in that last code modification he wrote '=' instead of '<=''. He fixes the bug, which causes T11 and T53 to be re-selected. Once these tests are re-executed, both tests pass.

## **Example tools**

The following tools can help Agile teams to find new bugs immediately:

- JUnitMax;
- Infinitest;
- Visual Studio – embedded.

### **3.4.6 Cloud computing and virtualisation tools**

Some types of tests (for example, performance and reliability tests) need more hardware resources than others. Some tests change test data in an irreversible manner, so test environments must be restored to an earlier state. In such situations, virtualisation tools and cloud computing tools come in handy.

Virtualisation tools can create snapshots of system states. When test environments must be recreated, the appropriate snapshot can be reloaded onto the virtualisation host. In this way, the state of the environment, including test data, is restored. Some test management tools now utilise virtualisation technologies to snapshot servers at the point when a failure occurs, allowing testers to share the snapshot with the developers investigating the underlying bug.

Other useful features of virtualisation and cloud computing are hardware cloning and provisioning new machines. When, for example, load generation tools need more resources, it is easy (provided one has enough real computing power) to clone existing machines and get more machines that are ready for use with already configured tools.

Service virtualisation is another technique that testers can use. It consists of emulating only the behaviour of the specific dependent components of the system that are needed to execute the tests. To avoid virtualising entire

systems, this technique allows specific fragments of a tested application to be emulated. This provides just enough application logic for the testers to implement and perform the tests even if the whole system is not ready yet.

## Example

In the gym chain scenario, some teams are developing software that does not run on the mainframe, but which must use services provided by the mainframe software. They can use a mainframe emulator to lower the costs and simplify the maintenance of the test environments.

One of the project goals is to improve disaster recovery by using cloud-based methods, so clouds will be part of the test environment. Testers can utilise clouds not only for deployment of tested software, but also for putting test automation infrastructure in the cloud. This makes administration and maintenance of test work products easier and makes test environments more robust. Thus, many project risks relating to tests can be mitigated.

## Example tools

The following tools can help Agile teams with virtualisation:

- Platform virtualisation:
  - HyperV;
  - Virtual Box;
  - Amazon Elastic Compute Cloud;
  - IBM Smart Cloud.
- Service virtualisation:
  - Parasoft Service Virtualisation;
  - IBM Service Virtualisation;
  - CA Service Virtualisation.

## Summary

In this section, we saw ways that tools can help testers deal with the competing demands of high quality standards, frequent delivery and rapid change. Regression tests support via test execution tools and continuous

integration tools is important. With some techniques, such as TDD, tool support is essential. While this section primarily looked at tools that are designed especially for use on Agile projects, we also recognised that basic testing tools, such as those described in the Foundation Level syllabus, are still applicable.

## Test your knowledge

The answers to these questions can be found in the Appendix.

### Agile Tester Sample Question – [Chapter 3 – Section 3.4](#)

#### Question 1 FA-3.4.1 (K1)

Can tools from traditional methodologies be used in Agile projects?

#### Answer set:

- A. Yes, testing tools are methodology-blind and can always be used in any project
- B. Yes, some of traditional testing tools can be used in Agile methodologies, but others are not compatible with them
- C. No, Agile teams prepare their own proprietary tools for every project
- D. No, traditional tools, such as test management tools are useless in Agile projects

## REFERENCES

- Adzic, G. (2009) *Bridging the Communication Gap: Specification by example and Agile acceptance testing*. Neuri Limited, London, UK.
- Beck, K. (2002) *Test-driven Development: By example*. Addison-Wesley Professional, Boston, MA, USA.
- Beizer, B. (2008) *Software Testing Techniques*. 2nd edn. Wiley & Sons, India.

- Black, R. (2014a) *Advanced Software Testing: Volume 1*. Rocky Nook, Santa Barbara, CA, USA.
- Black, R. (2014b) *Advanced Software Testing: Volume 2*. Rocky Nook, Santa Barbara, CA, USA.
- Black, R., Graham, D. and van Veenendaal, E. (2012) *Foundations of Software Testing: ISTQB certification*. Cengage Learning, Andover, UK.
- Chelimsky, D., Astels, D., Dennis, Z., Hellesøy, A., Helmckamp, B. and North D. (2010) *The RSpec Book: Behaviour-driven development with RSpec, Cucumber and friends*. Pragmatic Bookshelf, Raleigh, NC, USA.
- Cohn, M. (2005) *Agile Estimating and Planning*. Mountain Goat Software, Prentice Hall, Upper Saddle River, NJ, USA.
- Coplien, J. (n.d.) Why Most Unit Testing is Waste. RBCS. Available at <http://rbcstech.com/resources/articles/why-most-unit-testing-is-waste/> (accessed April 2017).
- Crispin, L. and Gregory, J. (2008) *Agile Testing: A practical guide for testers and Agile teams*. Addison-Wesley Professional, Boston, MA, USA.
- Gerrard, P. (2009) *The Tester's Pocketbook*. The Tester's Press, Berkshire, UK.
- Gregory, J. and Crispin, L. (2014) *More Agile Testing: Learning journeys for the whole team*. Addison-Wesley Professional, Boston, MA, USA.
- Hambling, B. Morgan, P. and Samaroo, A. (2010) *Software Testing: An ISTQB-ISEB foundation guide*. BCS, Swindon, UK.
- Hendrickson, E. (2013) *Explore It! Reduce risk and increase confidence with exploratory testing*. Pragmatic Bookshelf, Raleigh, NC, USA.
- Howden, W. E. (1976) Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering* SE-2(3): 208–215. Available at <http://ieeexplore.ieee.org/document/1702367/> (accessed April 2017).
- Kramer, A. and Legeard, B. (2016) *Model-based Testing Essentials. Guide to the ISTQB certified model-based tester: foundation level*. Wiley & Sons, Hoboken, NJ, USA.
- Lehtonen, T., Tuomivaara, S., Rantala, V., Känsälä, M., Mäkilä, T., et al. (2014) *Sulautetujen järjestelmien ketterä käzikirja (Agile Handbook for Embedded Systems)*. Turku: Painosalama. Available at <http://trc.utu.fi/embedded/kasikirja> (accessed April 2017).
- Linz, T. (2014) *Testing in Scrum: A guide for software quality assurance in the Agile world*. Rocky Nook, Santa Barbara, CA, USA.
- Saff, D. and Ernst, M. D. (2004) Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)* (Barcelona, Spain), 30 March 2004. Available at

<http://groups.csail.mit.edu/pag/pubs/conttest-plugin-etc2004-abstract.xhtml> (accessed April 2017).

Schwaber, K. (2007) *The enterprise and Scrum*. Microsoft Press, Redmond, WA, USA.

van der Aalst, L. and Davis, C. (2013) *TMap NEXT® in Scrum*. [ICT-Books.com](http://ICT-Books.com).

Whitaker, J. A. (2002) *How to Break Software: A practical guide to testing*. Pearson, UK.

Whitaker, J. A. (2009) *Exploratory Software Testing: Tips, tricks, tours and techniques to guide test design*. Addison-Wesley Professional, Boston, MA, USA.

Whitaker, J. A. and Thompson, H. H. (2003) *How to Break Software Security*. Pearson, UK.

## FURTHER READING

Black, R. (2007) *Pragmatic Software Testing*. John Wiley & Sons, UK.

Black, R. (2010) Selection of Test Design Techniques. RBCS. Available at <http://rbcs-us.com/blog/selection-of-test-design-techniques-in-risk-based-testing/> (accessed April 2017).

Black, R., van der Aalst, L. and Rommens, J. L. (2017) *Expert Testing Manager*. Rocky Nook, Santa Barbara, CA, USA.

Cohn, M. (2004) *User Stories Applied: For Agile software development*. Addison-Wesley Professional, Boston, MA, USA.

Hendrickson, E. (2008) Acceptance Test-driven Development: An overview. Test Obsessed. Available at <http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview> (accessed April 2017).

Jones, C. and Bonsignour, O. (2011) *The Economics of Software Quality*. Addison-Wesley Professional, Boston, MA, USA.

van Veenendaal, E. (2012) *The PRISMA Approach*. Uitgeverij Tutein Nolthenius Publishers, the Netherlands.

## WEBSITES

ISTQB Foundation Working Group [www.istqb.org/downloads/category/2-foundation-level-documents.xhtml](http://www.istqb.org/downloads/category/2-foundation-level-documents.xhtml)

## NOTES

- 1 You can find the syllabus and other associated documents at [www.istqb.org/downloads/category/2-foundation-level-documents.xhtml](http://www.istqb.org/downloads/category/2-foundation-level-documents.xhtml) (accessed April 2017).
- 2 This concept is not unique or new to Agile development. Boris Beizer (2008) talks about the value of a test-first approach to software development.
- 3 You can find the initial description in Beck (2002).
- 4 One widely read source on the topic is Adzic (2009).
- 5 A good discussion on this topic is Chelimsky et al. (2010).
- 6 If you are not sure what these techniques are, see this article: [www.softwaretestingclass.com/boundary-value-analysis-and-equivalence-class-partitioning-with-simple-example/](http://www.softwaretestingclass.com/boundary-value-analysis-and-equivalence-class-partitioning-with-simple-example/) (accessed April 2017).
- 7 [www.duncannisbet.co.uk/dissecting-the-testing-quadrants-different-representations](http://www.duncannisbet.co.uk/dissecting-the-testing-quadrants-different-representations) (accessed August 2016).
- 8 <https://gojko.net/2013/10/21/lets-break-the-Agile-testing-quadrants/> (accessed August 2016).
- 9 You can find the proof of this unfortunate pudding in Howden (1976).
- 10 The concepts of test design techniques and their respective relevant coverage criteria, both white box and black box, are discussed in the Foundation Level syllabus as well as books that describe it, such as Black et al. (2012) and Hambling et al. (2010).
- 11 Delphic Oracle and other traditional estimation techniques are explained in Black (2014b).
- 12 For example, on the RBCS website, you will find lists of quality risk categories. You can download those lists, customise them and use them as checklists when you are identifying risk items: <http://rbcs-us.com/resources/templates-and-examples/> (accessed April 2017).
- 13 You can find a more detailed discussion of the allocation of test effort based on risk levels at <http://rbcs-us.com/blog/selection-of-test-design-techniques-in-risk-based-testing/> (accessed April 2017).
- 14 If you would like a more detailed description of this template and the risk analysis process, you can find it in Black (2014b). There are also templates, articles and videos on risk-based testing available at [www.rbcs-us.com](http://www.rbcs-us.com) (accessed April 2017).
- 15 [www.tmap.net/wiki/risk-poker](http://www.tmap.net/wiki/risk-poker) (accessed April 2017).

- 16 The normal or Gaussian distribution is also referred to as a bell curve, where the area of the curve is evenly and symmetrically distributed around the average. Many natural and human phenomena follow a normal distribution. As you can see in the discussion here, [www.itl.nist.gov/div898/handbook/eda/section3/eda3661.htm](http://www.itl.nist.gov/div898/handbook/eda/section3/eda3661.htm) (accessed May 2017), the normal distribution tends to apply in many situations, but small samples may exhibit deviation from it.
- 17 Delphic Oracle and other traditional estimation techniques are explained in Black (2014b).
- 18 A Fibonacci series is one where the first two terms are 1, and each subsequent term is the sum of the preceding two terms. Some variants define the first two terms as 0 and 1, but that would not make sense the way we are using the series here. As with the normal distribution discussed earlier, the Fibonacci series occurs widely in natural situations.
- 19 Options for studying black-box test design techniques include Black et al. (2012) and Hambling et al. (2010). In addition, at the Advanced level, you can refer to Black (2014a).
- 20 These techniques are described in the ISTQB® Foundation Level syllabus and books that describe it, such as Hambling et al. (2010) and Black et al. (2012).
- 21 For an explanation of Gherkin syntax see <https://docs.hiptest.net/writing-scenarios-with-gherkin-syntax/> (accessed April 2017).
- 22 Whitaker's attack techniques are discussed in his books *How to Break Software* (Whitaker, 2002) and *How to Break Software Security* (Whitaker and Thompson, 2003). Also relevant to this section is his book *Exploratory Software Testing* (Whitaker, 2009). Another resource is Hendrickson's *Explore It!* (2013).
- 23 To find detailed discussions of these strategies in books, see Black (2014b) and Black et al. (2012).
- 24 In addition to the syllabus itself, sources on Foundation Level topics include *Software Testing* (Hambling et al., 2010) and *Foundations of Software Testing* (Black et al., 2012).

## 4 APPENDIX: TEST YOUR KNOWLEDGE ANSWERS

### CHAPTER 1

#### Section 1.1: Agile tester sample questions

Question 1 correct answer: **C**

Justification:

A is **incorrect** because Agile projects attempt to develop only the solution required and do not try to ‘futureproof’ the system. Answer A does not address this as it is concerned with ensuring the right team is properly equipped to do the job.

B is **incorrect** because Agile projects attempt to develop only the solution required and do not try to ‘futureproof’ the system. Answer B does not address this as it focuses on ensuring the team is not overworked.

C is **correct** because simplicity is key in Agile projects and the emphasis is on the simplest solution possible.

D is **incorrect** because Agile projects attempt to develop only the solution required and do not try to ‘futureproof’ the system.

Answer D does not address this as it refers to how the team is organised rather than achieving simple design.

### Question 2 correct answer: C

Justification:

- i. is **true** as the ideal Agile team size is between three and nine.
- ii. is **false** as all members should attend daily stand-up meetings to report on what they have/have not achieved since the last stand-up meeting.
- iii. is **true** as all user story discussions should include developers, testers and business representatives (the ‘power of three’).
- iv. is **false** as the whole team should be co-located and integrated.

A is **incorrect** because (i) is true and (ii) is false.

B is **incorrect** because (iii) is true and (iv) is false.

C is **correct** because (i) and (iii) are both true.

D is **incorrect** because (ii) and (iv) are both false.

### Question 3 correct answer: A

Justification:

A is **correct** because the use of short development iterations means the customer gets working software early in the project and can then provide feedback.

B is **incorrect** because the ‘V’ model is a traditional software development model and is not used on Agile projects.

C is **incorrect** because Agile development thrives on unclear and changing requirements and is used where requirements are not clearly defined in advance.

D is **incorrect** because the use of web conferencing tools will have little or no impact on how quickly working software is developed.

## Section 1.2: Agile tester sample questions

Question 1 correct answer: **B**

Justification:

A is **incorrect** because XP means Extreme Programming. Experienced Processes is not a known Agile process.

B is **correct** because Scrum is organisational and XP brings development methods.

C is **incorrect** because Kanban is not related to scaled Agile teams.

D is **incorrect** because Scrum is primarily used to develop new features.

Question 2 correct answer: **D**

Justification:

User stories are the basis for development. At first the added value of the features for the customer has to be identified because here only technical issues have been discussed.

A is **incorrect** because we do not know if the current prototypes/developments reflect the user needs.

B is **incorrect** because if doing a product risk analysis is a good idea, it must be done with user representatives, developers and the product owner, and not only between a tester and the business representatives.

C is **incorrect** because we do not know if the current prototypes/developments reflect the user needs.

D is **correct** because this way of doing things will allow several people with different perspectives to discuss and understand what the product must do.

Question 3 correct answer: **C**

Justification:

- A is **incorrect** because this meeting is solution oriented.
- B is **incorrect** because improvements can concern processes, people and products.
- C is **correct** because it is factual and solution oriented.
- D is **incorrect** because the retrospective is dedicated to making decisions about improvements.

Question 4 correct answer: **B**

Justification:

- A is **incorrect** because it is a system, not a meeting where stories are added in the product backlog.
- B is **correct**. Static analysis tools report metrics. Other reports inform the developer on several topics such as building and compiling.
- C is **incorrect** because CI is not a test approach.
- D is **incorrect** because the developer has their own tools on their computer and putting the code in the configuration management system is mandatory for the CI to extract and build the code.

Question 5 correct answer: **C**

Justification:

- A is **incorrect** because an increment is linked to an iteration, not a version.
- B is **incorrect** because an iteration meeting focuses on tasks to be performed by the team.
- C is **correct** because release planning uses roughly estimated items, while iteration planning requires much greater estimation detail.

D is **incorrect** because, during the release meeting, the team has not completely thought about how to implement the backlog item.

## CHAPTER 2

### Section 2.1: Agile tester sample questions

Question 1 correct answer: **C**

Justification:

A is **incorrect**: testers on traditional teams also create and execute automated tests.

B is **incorrect**: testers on traditional teams should also collaborate with developers and business stakeholders.

C is **correct**: in traditional projects, the only test activity that occurs when development commences is test case design; in Agile projects, testers start testing when developers start coding and developing features.

D is **incorrect**: testers on traditional teams can also incorporate exploratory testing into their test activities.

Question 2 correct answer: **B** and **D**

Justification:

A is **incorrect**: the decision of which stories, features and tasks to be worked on in the next iteration is a team effort, and is decided during release and iteration planning sessions.

B is **correct**: defining the user acceptance criteria is done collaboratively with business stakeholders, developers and testers to ensure all parties understand the intent of the story, which also assists in removing any ambiguities.

C is **incorrect**: developers are still responsible for unit testing in

Agile projects; however, a tester may have input into the unit tests during pairing (see answer D).

D is **correct**: this generally happens during iteration planning sessions, where testers discuss testing tasks related to the planned user stories, and then throughout the iteration in more detail, especially with the use of BDD and ATDD.

E is **incorrect**: developers may assist the testers with the test automation by building hooks into the code, or developing modules in the test automation tool for testers to use; however, test automation can be done by anyone on the team, and is not just the responsibility of the developers.

Question 3 correct answer: **D**

Justification:

A is **incorrect**: as Agile has evolved, it has become evident that people with specific skillsets have specific roles on Agile teams, testing being one of these.

B is **incorrect**: this is sometimes the case, but is not true in every situation; it depends on the team's needs.

C is **incorrect**: generally independent testers can find more defects than developers, but this depends on the level of testing being performed and also the expertise of the independent tester.

D is **correct**: as discussed in the syllabus and in this book, the level and independence of testing required depends on the team's needs.

## Section 2.2: Agile tester sample questions

Question 1 correct answer: **C**

Justification:

A is **incorrect**: this is a valid response to the 'What is getting in

your way?’ question.

B is **incorrect**: this is a valid response to the ‘What have you completed since the last meeting?’ question.

C is **correct**: stand-up meetings should be short (ideally kept under 15 minutes), so analysis of obstacles should be done outside the meeting, not during the meeting.

D is **incorrect**: this is a valid response to the ‘What do you plan to complete by the next meeting?’ question.

Question 2 correct answer: **B**

Justification:

A is **incorrect**: while automation is critical in Agile development, this is not the best reason to review test cases from previous iterations. Test automation should only be updated once test design changes.

B is **correct**: Agile projects embrace changes, so everything from user stories to test cases may change and testers should be ready to update tests.

C is **incorrect**: this would not be a reason to set time aside in a new iteration to review test cases from a previous iteration.

D is **incorrect**: there is plenty to do for testers from the beginning of the iteration, such as writing acceptance test cases, test data design and preparation; above all, Agile teams should be self-organising, and testers can assume other team roles.

### Section 2.3: Agile tester sample questions

Question 1 correct answer: **B**

Justification:

A is **incorrect**: testers **should** have some development skills to

assist them with their tasks on an Agile team, but it is not mandatory (a **must**). Modern Agile practices recognise that people have specialist skillsets to perform roles within the team; however, some team members are multi-skilled and cross-functional to enable them to perform other roles within the team.

B is **correct**: all Agile team members should be results oriented. Testers should also be proactive and participate in all team activities to ensure the whole-team approach is applied.

C is **incorrect**: again, testers **should** be technically oriented to be able to automate tests, but it is not mandatory (a **must**) as some teams are set up to accommodate both technical and non-technical testers.

D is **incorrect**: testers should not **only** be business oriented. As mentioned above, it is recommended that testers have some technical or development skills to be able to assist the team and enable them to participate in technical activities on the team, such as pairing with developers, writing automated tests and so forth.

Question 2 correct answer: **C**

Justification:

A is **incorrect**: testers still participate in the definition of the test strategy when working in Agile teams.

B is **incorrect**: testers still write test cases on Agile teams to the required levels based on the overall project and product risk of the project.

C is **correct**: maintaining the iteration backlog is not generally the responsibility of a tester on an Agile team (unless they have been appointed that role). The iteration backlog contains detailed tasks and is not a low-level activity.

D is **incorrect**: testers are responsible for organising and

conducting their own work, including exploratory test activities.

## CHAPTER 3

### Section 3.1: Agile tester sample questions

Question 1 correct answer: **A**

Justification:

A is **correct**. All can be utilised by a developer, but TDD is integrally a developer methodology.

B is **incorrect**. BDD is a collaborative approach to acceptance test level.

C is **incorrect**. Domain analysis is a test design technique.

D is **incorrect**. ATDD is a collaborative approach to acceptance test level.

Question 2 correct answer: Answer: **B**

Justification:

A is **incorrect**. This is wrong as unit tests are done via API-type tools.

B is **correct**. All test pyramid levels could be automated, although they are not necessarily always automated.

C is **incorrect**. This is wrong as some or even all acceptance tests could be automated.

D is **incorrect**. This is wrong as it is the lower levels (unit testing) of the pyramid where it is cheaper to find and fix faults.

Question 3 correct answer: **A**

Justification:

A is **correct**. Usability tests belong to testing quadrant Q3, which is

business facing and critiques the product.

B is **incorrect**. This combination is not true, as described in [Figure 3.4](#).

C is **incorrect**. This combination is not true, as described in [Figure 3.4](#).

D is **incorrect**. This combination is not true, as described in [Figure 3.4](#).

Question 4 correct answer: **B**

Justification:

A is **incorrect**. This is wrong because any one technique does not work for all tasks, even if it is a good technique.

B is **correct**. Even if teams do not organise a specific Sprint Zero for a preparation task, they still need to prepare many things, such as tools and task boards.

C is **incorrect**. This is clearly wrong as the team should plan things together, as a whole team.

D is **incorrect**. This is wrong as you also need continuous integration if you are to use continuous testing. Otherwise, it might be a very good idea to test new functionalities as soon as they are ready on a continuous basis.

### Section 3.1: Agile tester exercises

Exercise 1 solutions:

Here are the **two** areas you should cover:

1. Guiding principles for the team:

- a. It appears that the team is in a need of change, so it would be good to remind them of the team practices of being open to feedback and being resilient.

- b. It appears that the team testing has been very unit testing focused, so they miss the whole picture of what testing can be. You could show and explain the Agile testing quadrants figure to the team.
  - c. It appears that the team members do not want to do more than unit testing. You need to convince them that the whole-team approach works and the team needs to be cross-functional.
2. Your own role in the team:
- a. As other team members for some reason have not come up with more testing than the unit testing, you should use your testing expertise and start with the highest priority testing activities that have been omitted, probably business-faced acceptance tests and exploratory tests.
  - b. As other team members have done unit test automation, the easiest step for them to help you in testing is probably to take up test automation of acceptance tests as well. You can guide them in the right direction. Be a quality coach.
  - c. You can also analyse what kind of defects came back from customers. These are probably the weak areas of the team (and its testing) and this information from an analysis will help you to decide what kind of testing is needed most.

## Section 3.2: Agile tester sample questions

Question 1 correct answer: **B**

Justification:

A is **incorrect**. The values given are story points not colours.

B is **correct**. Red is the consensus risk opinion.

C is **incorrect**. Since this is the first iteration, the team should discuss and re-vote.

D is **incorrect**. Since this is the first iteration, the team should

discuss and re-vote.

Question 2 correct answer: **D**

Justification:

A is **incorrect**. The smallest value is not selected when a deadlock occurs.

B is **incorrect**. The time limit has been reached.

C is **incorrect**. The time limit has been reached.

D is **correct**. Since there is no possibility for more iterations, the story point has to be the highest score given by the estimators, and in the case of a high score such as 40, the story has to be broken down into smaller ones.

### **Section 3.2: Agile tester exercises**

Exercise 1 solution:

The scrum master checks whether the time limit has been reached. If it has, the choice has to be made and, in this case, it is red, since the most risky estimation should be selected. If not, then it is time for another iteration. Since after the first iteration the high and low estimators defended their scores, it is reasonable that the estimators who changed their estimation defend the cause of their modifications.

Exercise 2 solution:

The moderator asks the tester and the business analyst to defend their highest/lowest estimation. After the defence speech is done, another iteration occurs, and the result will be checked.

### **Section 3.3: Agile tester sample questions**

Question 1 correct answer: **D**

Justification:

A is **incorrect**. This is testable and relevant, but not specific to this user story.

B is **incorrect**. This statement is relevant, but not testable.

C is **incorrect**. This statement is relevant, testable and specific, but it contradicts the intention of the user story itself, which only removes certain privileges, not whole accounts.

D is **correct**. This statement is relevant, testable, specific to and consistent with the user story.

Question 2 correct answer: **C**

Justification:

A is **incorrect**. This is a testable criterion, but it concerns reliability, not performance.

B is **incorrect**. While this is about performance and is testable, it does not address the entire report.

C is **correct**. This is testable and has to do with the entire report.

D is **incorrect**. The criterion is still not testable, and the issue of localisation has been introduced.

Question 3 correct answer: **A**

Justification:

A is **correct**. Since you can assume that all the other actions were already tested and the test data retained, this test re-uses that data.

B is **incorrect**. Since you can assume that all the other actions were already tested and the test data retained, there is no need for the first step in this test.

C is **incorrect**. Since you can assume that all the other actions were already tested and the test data retained, there is no need for the

first two steps in this test.

D is **incorrect**. Since you can assume that all the other actions were already tested and the test data retained, there is no need for the first three steps in this test.

Question 4 correct answer: **B**

Justification:

A is **incorrect**. As shown in [Figure 4.1](#), there are three partitions that need to be covered: account does not exist; account exists but has no view sales privileges to remove; and account exists and has sales privileges.

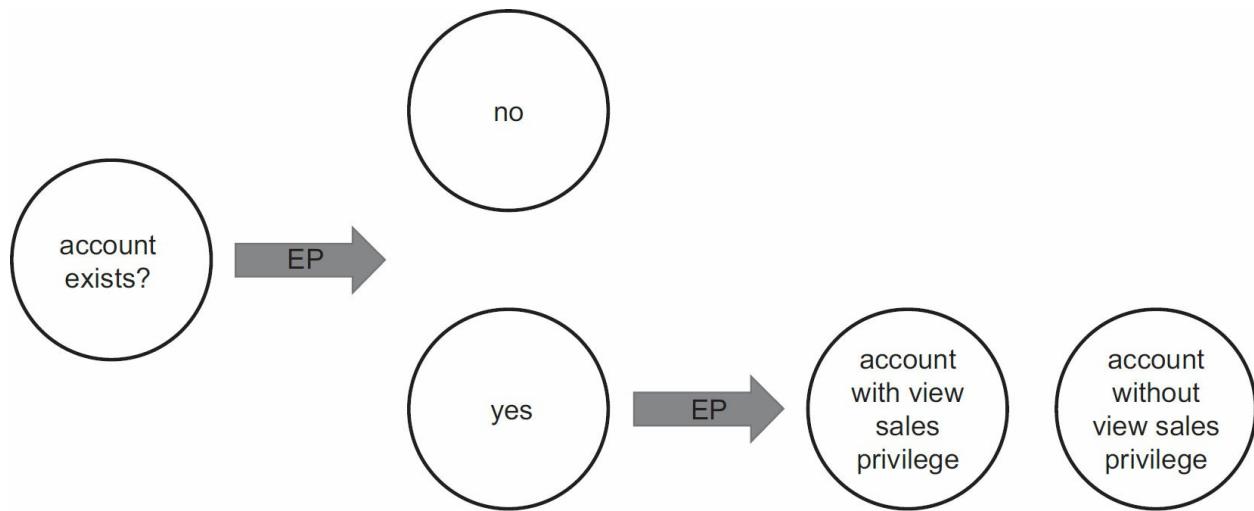
B is **correct**. As shown in [Figure 4.1](#), there are three partitions that need to be covered: account does not exist; account exists but has no view sales privileges to remove; and account exists and has sales privileges.

C is **incorrect**. As shown in [Figure 4.1](#), there are three partitions that need to be covered: account does not exist; account exists but has no view sales privileges to remove; and account exists and has sales privileges.

D is **incorrect**. As shown in [Figure 4.1](#), there are three partitions that need to be covered: account does not exist; account exists but has no view sales privileges to remove; and account exists and has sales privileges.

---

**Figure 4.1 Equivalence partitions for the account test**



Question 5 correct answer: **C**

Justification:

A is **incorrect**. One or more test conditions define the scope of the charter. Relative priority helps people to decide the order in which to run the exploratory tests. Pre-loaded test data files can make the tests more efficient. Test bases documents such as requirements and user stories give test ideas. The test oracle allows determination of the expected result. However, test steps and expected results do not belong, as these constrain the test and make it not exploratory, but rather scripted.

B is **incorrect**. One or more test conditions define the scope of the charter. Relative priority helps people to decide the order in which to run the exploratory tests. Pre-loaded test data files can make the tests more efficient. Test bases documents such as requirements and user stories give test ideas. The test oracle allows determination of the expected result. However, test steps and expected results do not belong, as these constrain the test and make it not exploratory, but rather scripted.

C is **correct**. One or more test conditions define the scope of the charter. Relative priority helps people to decide the order in which

to run the exploratory tests. Pre-loaded test data files can make the tests more efficient. Test bases documents such as requirements and user stories give test ideas. The test oracle allows determination of the expected result. However, test steps and expected results do not belong, as these constrain the test and make it not exploratory, but rather scripted.

D is **incorrect**. One or more test conditions define the scope of the charter. Relative priority helps people to decide the order in which to run the exploratory tests. Pre-loaded test data files can make the tests more efficient. Test bases documents such as requirements and user stories give test ideas. The test oracle allows determination of the expected result. However, test steps and expected results do not belong, as these constrain the test and make it not exploratory, but rather scripted.

### **Section 3.3: Agile tester exercises**

Exercise 1 solution:

The criteria can include both general and specific criteria, but should focus on specific criteria. Examples of specific criteria include:

The city should default to the current location but allow the user to select or type in another city.

During list creation, the app should suggest restaurants that the user has previously reviewed.

Guest accounts should not be able to create a list of favourite restaurants, but should be able to see that the feature exists, so that, if they try to create a list, they are invited to register.

The user interface should conform to the look-and-feel of the entire app.

List creation should start within 0.5 seconds of the user tapping the

‘create favourites’ button.

Other criteria are possible.

### Exercise 2 solution:

You should have used one or more of the following techniques from the Foundation syllabus: equivalence partitioning, boundary value analysis, decision tables, state transition diagrams or use cases. The specific technique(s) is determined by the criteria you wrote down. As an example, consider this criterion:

The city should default to the current location but allow the user to select or type in another city.

This would be a good candidate for state-based testing, since the way the process works depends on whether the user accepts the default. If so, then the city selection is done. If not, then the user either selects from a list or types a city name in directly.

As discussed in the text, the process of creating these tests would likely lead to questions for the team. In this example, some additional information about the screens and the navigation flow would be needed to define tests.

### Exercise 3 solution:

Here again, the specific answer depends on the acceptance criteria you created and which of them you decided to test. For example, if you created tests for the city selection criterion, your table could look as shown in [Table 4.1](#) (only partially completed).

---

**Table 4.1 Example of ATDD solution**

ID	Action	Expected Result
1	Tap 'create favourites'	Show current city
2	Click 'OK'	Move to 'add favourites' dialog
3	Select first restaurant	Restaurant name put on favourites list
4	Click 'save list'	Return to home screen
5	Tap 'create favourites'	Show current city
6	Click 'OK'	Error message 'city already exists'
7	Click 'change'	Display 'select or edit' message
8	Click 'edit'	Display free-form text field
9	Input 'Test City 1'	'Test City 1' displayed as entered
10	Click 'create'	Move to 'add favourites' dialog

#### Exercise 4 solution:

Once again, your specific answer depends on the criteria you defined and which one(s) you selected for this exercise. At the very least, your charter should define the test condition(s) to cover and the time limit (i.e. the time box).

You might also describe the actor, the intended user of the system whom you are representing during this exploratory test session, along with the user's persona. If preconditions or data must be created prior to the test, a setup section can describe that and a data section might be used if data must be available during the test. It is a good idea, when multiple charters are defined, to have a relative priority based on the priority of the associated user story or the risk level (defined during the risk analysis discussed earlier in this chapter). You can have a reference section for specifications, risks, test basis documents and other information sources. You can also have a separate section on the oracle, especially for features where the correct behaviour may be subject to interpretation.

Finally, some people would include a list of activities that the user is likely to

want to do or that would be interesting to try, along with some variations, settings changes and other ideas. However, keep in mind that the more of this kind of information you define up front, the greater the risk that the test charter changes from an exploratory test into more of a logical test case.

### **Section 3.4: Agile tester sample questions**

Question 1 correct answer: **B**

Justification:

A is **incorrect**. Some tools support activities that are not present in Agile projects (scheduling for example), so they will not be used in an Agile environment.

B is **correct**. Such partial applicability is explained in [Section 3.4](#) of Foundation Extension Level Syllabus – Agile Tester.

C is **incorrect**. Agile teams use proprietary tools, but use also open-source and commercial tools.

D is **incorrect**. The possible use of such tools is explained in [Section 3.4](#) of Foundation Extension Level Syllabus – Agile Tester.

# **5 AGILE TESTER SAMPLE EXAM**

## **Questions**

### **Question 1**

According to the Agile Manifesto's statements of values, rather than a reliance on tools and processes, Agile projects value which of the following:

#### **Answer set:**

- A. The presence of a clear contract of work agreed with the customer before starting development
- B. Communication and interaction between team members at all stages of the development cycle
- C. The creation of supporting user documentation during each development iteration
- D. Having a detailed plan in advance of each system release that is closely followed during the release

### **Question 2**

According to the Agile Manifesto which of the following is valued more highly than 'comprehensive documentation'?

**Answer set:**

- A. Simple plans
- B. User stories
- C. Working software
- D. Automated builds

**Question 3**

Agile development encourages a whole-team approach to software development. With regard to the whole-team approach, which of the following statements are true and which are false?

- i. Testers' expertise means they have ultimate responsibility for quality
- ii. Developers should be encouraged to talk directly to business representatives where they are unclear about requirements
- iii. Testers should consult developers when writing automated tests
- iv. Business representatives should write acceptance tests independently from the rest of the team
- v. Business representatives should have no role in discussions where the development team is estimating the size of user stories

**Answer set:**

- A. (i) (iii) and (iv) are true; (ii) and (v) are false
- B. (ii) and (iii) are true; (i), (iv) and (v) are false
- C. (iv) and (v) are true; (i), (ii) and (iii) are false
- D. (ii), (iv) and (v) are true; (i) and (iii) are false

**Question 4**

Which of the following is an advantage of having the whole team responsible for quality?

**Answer set:**

- A. Testers are now responsible for quality across the whole project, so developers can concentrate on writing code
- B. Developers are responsible for writing automated tests and keeping them up to date in the continuous integration build server so testers can focus on exploratory testing activities
- C. All team members are skilled in all roles on the Agile team, so any other team member can easily take over the work of another team member if they are ill
- D. Role boundaries are broken down and team members contribute to project success by leveraging their unique skillsets and perspectives

**Question 5**

Which of the following BEST describes a benefit associated with early feedback?

**Answer set:**

- A. Customers do not have to write acceptance tests as they are constantly providing feedback to the development team
- B. Testers spend less time overall writing tests as they are in regular contact with customers
- C. Defect density increases because developers better understand what the customer wants
- D. Using continuous integration, quality problems can be highlighted earlier when they are easier to fix

**Question 6**

Which of the following statements are true?

- i. Early feedback gives the developers more time to develop system features

- ii. Early feedback gives the team confidence that the requirements are correct
- iii. Early feedback reduces the amount of time needed for system testing
- iv. Early feedback reduces re-work in later iterations of the project

**Answer set:**

- A. (i) and (iv) are true; (ii) and (iii) are false
- B. (ii) and (iii) are true; (i) and (iv) are false
- C. (ii) and (iv) are true; (i) and (iii) are false
- D. (i) and (iii) are true; (ii) and (iv) are false

**Question 7**

Which of the following statements is true about Kanban, Scrum and XP?

**Answer set:**

- A. Kanban is mainly used to transition from XP to Scrum
- B. Both XP and Scrum aim to propose good practices for development
- C. Scrum proposes a framework for team organisation and project management
- D. XP and Kanban are based on a management flow

**Question 8**

Consider an Agile project to develop a website aimed at selling goods. Statistics about the sales of the day may be extracted. Consider the following user story from this Agile project:

As a sales manager

I want to be able to save the daily sales report

So that I can compare it with the next day's sales

For this user story the following acceptance criteria are defined:

The sales manager must receive a daily sales report.

The sales manager can save the sales report when they click the ‘Save’ button.

Based only on the given information, which of the following is MOST LIKELY to be true?

**Answer set:**

- A. A conversation between the tester, developer and product owner could highlight the need to select a place where to save the report
- B. During a conversation between the tester, developer and product owner, the acceptance criteria could be changed to be user-oriented instead of GUI-oriented
- C. The user story should be replaced by the acceptance criteria, and the acceptance criteria by the story
- D. The user story and its acceptance criteria are sufficiently detailed

**Question 9**

You are a tester working on an Agile project that is building a retail stock control system. The business owner has defined the following user story:

As a user

I want to be able to manage stock codes

So that I can remove any that are obsolete.

Based on the above, what would be your next task?

**Answer set:**

- A. The user story is constructed correctly, so you would start defining test cases for the user story
- B. The user story is constructed correctly, so you work with the developer and business owner to define the acceptance criteria

- C. The user story is not constructed correctly, so you work with the business owner to put it in the correct format of Given- When-Then
- D. The user story is not constructed correctly, so you work with the business owner to clarify the conditions

### **Question 10**

Which of the following statements is true within an Agile team?

#### **Answer set:**

- A. Retrospectives allow the team to get experience from other successful Agile projects
- B. All relevant stakeholders should attend the retrospective if they affect the team activities
- C. During the retrospective, the demonstration provides feedback on the product capabilities
- D. Retrospectives aim to conclude the project and bring the team together using Agile games

### **Question 11**

Which of the following BEST describes the testers' involvement in a retrospective meeting?

#### **Answer set:**

- A. Testers should only attend the retrospective if they have something to contribute
- B. Testers should only attend the retrospective as observers
- C. Testers should only discuss topics related to testing during the retrospective meeting
- D. Testers should discuss any topic related to the team during the retrospective meeting

## **Question 12**

Which of the following BEST represents how a continuous integration system (CIS) is used in Agile projects?

### **Answer set:**

- A. A CIS helps to automatically generate code from a model
- B. The CIS builds and checks the quality of the whole-team developments
- C. The goal of the CIS is to deploy builds in production
- D. The CIS is a development framework installed on each developer's station

## **Question 13**

Which of the following statements is true within an Agile team?

### **Answer set:**

- A. Release planning meetings refine all the testing tasks to be performed for a release
- B. Iteration planning meetings define the tasks to be performed by a tester during an iteration
- C. Testers are involved in iteration planning, not in release planning
- D. Release planning meetings define testing tasks to be conducted during iteration planning meetings

## **Question 14**

Consider the following statements and determine which are true and which are false statements about Agile testing activities:

- i. Business stakeholders are involved in testing activities
- ii. Test automation is mandatory in Agile development
- iii. System test documentation is not produced in Agile development

iv. Quality is the responsibility of everyone on the Agile team

**Answer set:**

- A. All statements are true
- B. (ii) and (iii) are true; (i) and (iv) are false
- C. (iii) and (iv) are true; (i) and (ii) are false
- D. (i) and (iv) are true; (ii) and (iii) are false

**Question 15**

You have just joined an Agile project team as a tester. Which one of the following is the main difference you expect on this project compared to a traditional project?

**Answer set:**

- A. You are expected to write unit tests that are used in the continuous integration build
- B. You are expected to automate all your system tests
- C. You are expected to contribute to acceptance criteria on user stories
- D. You are expected to write the user stories

**Question 16**

Which of the following team structures within an Agile project would best demonstrate independence of testing while maintaining the whole-team approach?

**Answer set:**

- A. You are the only tester embedded in an Agile team, where the developers write and automate the majority of the tests
- B. You are a tester in a separate independent test team, where you and other testers are assigned to an Agile project on a long-term basis
- C. You are a tester in a separate independent test team, where you and

- other testers are assigned to different Agile projects on an on-demand basis at the end of each sprint
- D. You are the only tester in the company, where you provide expert testing consultancy to developers and business representatives across multiple Agile projects

### **Question 17**

Jane is a tester at a large financial institution. She is part of the company's test team, and is currently assigned on a long-term basis to an Agile project delivering a new internet banking portal. Jane has identified that performance testing and security testing are required on the project.

Based on Jane's company and team structure, how should Jane address the need for performance and security testing?

#### **Answer set:**

- A. Jane should be able to perform this testing as part of her role as a tester on the Agile team
- B. Jane should request the developers to perform this testing as they are more familiar with the code
- C. Jane should seek the assistance from specialists in these fields within the company's test team to perform this testing
- D. Jane should request the company to hire specialist outsourced consultants to perform this testing

### **Question 18**

The product owner for the Agile project you are working on has asked to see the status of testing for the current iteration. Which of the following would best provide the product owner with this information?

#### **Answer set:**

- The test management tool, which the test team updates at the end of
- A. each iteration
  - B. The defect management tool, which the team updates when a defect is found or resolved
  - C. The project story board, which the team continuously updates when tasks are updated
  - D. The team burndown chart, which is updated immediately when tasks are updated

### **Question 19**

As a tester in an Agile team, you have encountered an issue with a lack of test data and misconfigurations in the test environments which are preventing you from verifying several user stories, therefore you are unable to progress. Of the following options, which is the BEST form of communicating these issues?

#### **Answer set:**

- A. Raise a defect and assign it to the developers of the user stories to address the issues
- B. Raise the issues on the daily test status report, and move on to verifying other user stories
- C. Raise the issues during the team stand-up meeting so that the whole team is aware of the issues and they can be addressed accordingly
- D. Raise the issues with the iteration manager so that they can find the required people to resolve the issues

### **Question 20**

Of the following, which is the MOST likely reason why regression risk on Agile projects needs a greater degree of attention than on traditional projects?

#### **Answer set:**

- A. Agile projects have a much larger scope of work than traditional projects
- B. Agile projects have a much larger code churn than traditional projects
- C. Agile project teams are smaller, and therefore have fewer people doing more work, which can introduce product regression
- D. Agile project teams use the latest cutting edge technologies, which may not be as mature as traditional technologies and which can introduce product regression

## **Question 21**

On an Agile project, at which test level should you automate test cases to help reduce the risk of product regression?

### **Answer set:**

- A. At the system test level
- B. At the integration test level
- C. At the unit test level
- D. At all test levels

## **Question 22**

Which of the following statements is true within an Agile team?

### **Answer set:**

- A. Testers are responsible for maintaining the quality of user stories
- B. Testers are responsible for ensuring test tools are utilised correctly by the team
- C. Testers are responsible for organising and managing iteration planning sessions
- D. Testers are responsible for maintaining the quality of unit tests

## **Question 23**

Which of the following statements represents the BEST definition of the skills of a tester in an Agile team?

### **Answer set:**

- A. As a team member in an Agile team, the tester is able to conduct any of the activities within the team
- B. As a tester in an Agile team, a tester needs to be able to execute all the testing tasks
- C. As a tester in an Agile team, the tester must not be involved in any developers' tasks
- D. As a tester in an Agile team, a tester is able to plan and organise their own work

## **Question 24**

Which TWO of the following activities would BEST describe the activities of a tester on an Agile team?

### **Answer set:**

- A. As a tester on an Agile project, you assist in coaching other team members on testing related activities
- B. As a tester on an Agile project, you participate in code review sessions, providing feedback on code quality
- C. As a tester on an Agile project, you participate in retrospective meetings, providing feedback on all activities within the project
- D. As a tester on an Agile project, you are responsible for defining the acceptance criteria for user stories
- E. As a tester on an Agile project, you assist the product owner with defining user stories

## **Question 25**

Which of the following would be used to define the objective of an exploratory testing session?

**Answer set:**

- A. A test scenario
- B. A session-based test statement
- C. A test charter
- D. A test oracle

**Question 26**

What is the typical format of a test generated using behaviour-driven development (BDD)?

**Answer set:**

- A. As a..., I want..., so that...
- B. Given..., When..., Then...
- C. In a scenario..., when behaviour..., do the following...
- D. If..., then..., else...

**Question 27**

According to the test pyramid, which test level should have the most test cases?

**Answer set:**

- A. Acceptance testing
- B. Integration testing
- C. Unit testing
- D. System testing

**Question 28**

Which of the following would be the most effective to use to ensure all

important test levels and test types are covered in an Agile project?

**Answer set:**

- A. The test pyramid
- B. The testing quadrants
- C. Test-driven development
- D. Behaviour-driven development

**Question 29**

Which of the testing quadrants includes user acceptance testing?

**Answer set:**

- A. Q1 – Technology-Facing, Supporting the Team
- B. Q2 – Business-Facing, Supporting the Team
- C. Q3 – Business-Facing, Critique Product
- D. Q4 – Technology-Facing, Critique Product

**Question 30**

You are part of an Agile team and you have just been to the sprint planning meeting where a number of user stories were chosen to be included in the team's sprint backlog. You now need to decide how you will design and execute your test cases. Which of these options would be your best choice?

**Answer set:**

- A. You will read through all user stories and design 10–20 test cases per user story and input them into the test management system, and in the last few days of the sprint you will then execute the test cases
- B. You will design detailed test cases based on the user stories that were chosen, so that you can start testing in the test environment after the sprint is over
- C. You will pair up with a developer and help them to design unit test

cases and execute them as they develop the code to be ready for unit testing

- D. You will read through the user story that is first on the coding priority of the developers, design a few test cases for it, and execute them as soon as the user story code is moved into the test environment

### Question 31

The risk table for three stories is the following:

	4	8	12	16
p r o b a b i l i t y	3	6	9	12
	2	4	6	8
	Story A		Story C	
	1	2	3	4

i m p a c t

Select the two test solutions that can be applied for this risk table.

#### Answer set:

- A. Story A: exploratory testing
- Story B: exploratory testing, defect prevention, model-based testing
- Story C: exploratory testing, model-based testing

- B. Story A: model-based testing, exploratory testing, defect prevention  
Story B: exploratory testing, defect prevention  
Story C: exploratory testing, defect prevention
- C. Story A: exploratory testing, defect prevention  
Story B: exploratory testing, defect prevention, model-based testing  
Story C: exploratory testing
- D. Story A: exploratory testing  
Story B: exploratory testing, static analysis, equivalence partitioning with boundary value testing  
Story C: exploratory testing, equivalence partitioning with boundary value testing
- E. Story A: exploratory testing, equivalence partitioning with boundary value testing  
Story B: exploratory testing, static analysis, equivalence partitioning with boundary value testing  
Story C: equivalence partitioning with boundary value testing

### **Question 32**

Take the following user story:

As the product owner, I need a sorting function where lower case and capital letters are not differentiated.

During the first iteration of the poker planning session, the following story points were given:

Scrum master: 8  
Developer: 5  
Business analyst: 13  
Tester: 8

What is the best outcome following this planning session?

**Answer set:**

- A. Since the scores are well balanced, there is no need for more iterations and the story point to be given is 8
- B. Since there is no consensus, an additional voting iteration is needed. Prior to the next voting, the business analyst and the developer have to defend their estimates since these two members gave the lowest and the highest values
- C. Since there is no consensus, an additional voting iteration is needed. Prior to the next voting, the Scrum master and the tester explain their estimations since theirs are in the middle and probably the most acceptable
- D. A discussion for consensus will follow, in which the Scrum master and the tester convince the two others to vote for 8

**Question 33**

As a tester on an Agile project developing an upgrade to the customer management and sales leads system, in addition to the user stories, which of the following would BEST provide relevant information to support your functional testing activities?

**Answer set:**

- A. You have a discussion with the customer on their expectations of how the system should perform during peak operations
- B. You review the defect analysis reports from similar products previously implemented
- C. You read the project plan and project schedule constructed at the commencement of the project
- D. You analyse all the project risks that were identified during project planning

**Question 34**

Given the following user story:

As a customer logged into an online store  
I want to be able to add a minimum of 1 item and up to 50 items to a personal wish list  
So that I can quickly locate these items at a later date when I'm ready to purchase them.

Consider the following acceptance criteria for testability

- i. I can add 1 to 50 items into my personal wish list
- ii. I will receive an error if I attempt to add more than 50 items to my personal wish list
- iii. I am able to easily locate items to add to my personal wish list
- iv. I can only add items to my personal wish when I am logged into the online store
- v. I can quickly retrieve and review my wish list items

**Answer set:**

- A. (i) (ii) and (iv) are testable; (iii) and (v) are not testable
- B. (i) (iii) and (v) are testable; (ii) and (iv) are not testable
- C. (ii) (iii) and (iv) are testable; (i), (ii) are not testable
- D. (ii), (iv) and (v) are testable; (i) and (iii) are not testable

**Question 35**

For the following user story:

As a conference manager  
I want up to 1000 customers to be able to register online concurrently  
So they receive their registration confirmation within 3 seconds of submitting their details.

The business representative has advised that an acceptance criterion for their

user story is ‘1000 concurrent users can register online and receive their registration confirmation within 3 seconds of submitting their details’.

Which of the following topics addresses the above acceptance criterion?

- A. Functional behaviour
- B. Quality characteristics
- C. Business rules
- D. Constraints

### **Question 36**

Given the following user story:

As a customer of an online flight booking system  
I want to save my credit card details  
So I can quickly book flights using my saved credit card details.

Which of the following is the BEST example of an acceptance test-driven development test case for this user story?

#### **Answer set:**

- A. Verify the credit card check-sum value is correct before committing the credit card number to the database
- B. Given I have logged into the system, when I enter my credit card details and click submit, then my credit card details are saved to my account
- C. Login to the flight booking system where you have previously saved your credit card details, make another flight booking and verify you are able to pay with your previously saved credit card details
- D. Login to the flight booking system where you have previously saved your credit card details, click on the My Account page, and verify you are able to change your credit card details

## **Question 37**

Given the following user story:

As an ATM operator

I want to give customers 3 attempts to enter their correct Personal Identification Number (PIN) to access their account

So that on the third incorrect attempt the ATM machine retains their ATM card.

Which of the following is the best black-box test design technique for the user story?

**Answer set:**

- A. Boundary value analysis. Test the following inputs: 0,1,3 PIN attempts
- B. State transition testing. Test the following states: S1 Start, S2 1st try, S3 2nd try, S4 3rd try, S5 Access to account, S6 Retain card
- C. Decision tables. Test the following conditions: User inserts card; At least 1st PIN attempt; PIN rejected; 2nd PIN attempt; PIN accepted with the resulting action of – Access account
- D. Use case testing: Actor=customer; Pre-requisites=customer inserts ATM card, enters PIN; Post-conditions=Access account

## **Question 38**

You have defined the following purpose in the test charter for your 120-minute exploratory testing session:

Explore the online flight booking system with various browsers and settings to discover defects related to unsupported browser configurations.

What would you expect to do during this exploratory test session?

**Answer set:**

- A. During the test session, you would survey and explore the various web browser configurations to include the details of these configurations in scripted test cases
- B. You would spend the first 30 minutes of the exploratory test session defining the test charter activities you will perform during the test session and the remaining time executing these activities using various web browsers and configurations
- C. Using the information contained in the test charter, you would commence parallel test design and test execution activities, recording your test results as you progress through the system using various web browsers and configurations
- D. Commence ad hoc testing of the online flight booking system using various web browsers and configurations to find as many defects as possible

**Question 39**

Tools that run processes of static verification, building, testing and deploying applications to designated environments are called:

**Answer set:**

- A. Continuous integration tools
- B. Automated test execution tools
- C. Data loading tools
- D. Exploratory testing tools

**Question 40**

Mind maps BEST support which TWO of the following test activities?

**Answer set:**

- A. Test design
- B. Automated test execution
- C. Exploratory testing
- D. Load test
- E. Code coverage measurement

## Answers

### Question 1

Correct answer: **B**

Justification:

A is **incorrect** because Agile projects value customer collaboration over contract negotiation.

B is **correct** because Agile projects value individuals and interactions over processes and tools.

C is **incorrect** because Agile projects value working software rather than comprehensive documentation.

D is **incorrect** because Agile projects value responding to change rather than following a predetermined plan.

### Question 2

Correct answer: **C**

Justification:

A is **incorrect** because the Agile Manifesto statement of value related to this question is ‘Working software *over* comprehensive documentation’.

B is **incorrect** for the same reasons as A.

C is **correct** because the Agile Manifesto statement of value related to this question is ‘Working software *over* comprehensive

documentation'.

D is **incorrect** for the same reasons as A.

### Question 3

Correct answer: **B**

Justification:

- i. is **false** as the whole team (not just testers) have responsibility for quality.
- ii. is **true** as collaboration between all members is an essential part of the whole-team approach.
- iii. is **true** as testers should collaborate with developers when writing automated tests.
- iv. is **false** as testers and business experts work together to write testable acceptance criteria.
- v. is **false** as the whole team, including business representatives, is involved in any meeting where features are being analysed or estimated.

A is **incorrect** because (i), (iv) and (v) are false and (ii) and (iii) are true.

B is **correct** because (ii) and (iii) are true and (i), (iv) and (v) are false.

C is **incorrect** because (i), (iv) and (v) are false and (ii) and (iii) are true.

D is **incorrect** because (i), (iv) and (v) are false and (ii) and (iii) are true.

### Question 4

Correct answer: **D**

Justification:

A is **incorrect** as quality is everybody's responsibility on Agile teams.

B is **incorrect** as developers are not responsible for all test automation; testers also need to do test automation on Agile teams.

C is **incorrect** as not all team members have the same skillsets in an Agile team. All team members can contribute to other roles, but Agile teams rely on leveraging unique skillsets in individual team members.

D is **correct** as the traditional role barriers (handover from development to test) is broken down, and testers, developers and business staff now work together collaboratively, contributing their unique skillsets and perspectives.

## Question 5

Correct answer: **D**

Justification:

A is **incorrect** because, in order to ensure features are done, testers and customers will collaborate to write acceptance tests.

B is **incorrect** because the time spent testing will remain at least the same, as testers now have more knowledge of the product and therefore what to test.

C is **incorrect** because defect density, if anything, should reduce as developers have a greater understanding of customer requirements.

D is **correct** because the use of continuous integration reveals any system integration issues early in the cycle when these issues are cheaper and easier to repair.

## Question 6

Correct answer: **C**

Justification:

- i. is **false** as early feedback ensures that the team is on the right track – it does not give more time to developers/testers.
- ii. is **true** as early feedback avoids requirements misunderstanding earlier, rather than at the end where it is more expensive to remediate.
- iii. is **false** as it does not reduce the amount of testing required; as with option (i) early feedback ensures the team is on the right track.
- iv. is **true** as with option (ii), having early feedback ensures that the team have understood the business requirements, and that the business have an early intervention point in case the system is not behaving the way they expect. Finding out later in the project can result in a very costly re-work effort.

A is **incorrect** because (i) and (iii) are false and (ii) and (iv) are true.

B is **incorrect** because (i) and (iii) are false and (ii) and (iv) are true.

C is **correct** because (i) and (iii) are false and (ii) and (iv) are true.

D is **incorrect** because (i) and (iii) are false and (ii) and (iv) are true.

## Question 7

Correct answer: **C**

Justification:

A is **incorrect** as transition is not addressed here. XP and Scrum are different and separate Agile processes.

B is **incorrect** as XP, but not Scrum, proposes development practices.

C is **correct** as Scrum provides a framework for organising teams and managing projects.

D is **incorrect** as only Kanban, not XP, uses a management flow.

## Question 8

Correct answer: **A**

Justification:

A is **correct** because, the conversation between the tester, developer and product owner would identify that the details of where to save the report have not been defined in the story.

B is **incorrect** because there is no need for the acceptance criteria to be user-oriented, only the user story should be user-oriented.

C is **incorrect** because the user story needs to be user-oriented, and the acceptance criteria should provide the actions to verify acceptance of the user story.

D is **incorrect** because the story is not sufficiently defined, due to the missing details identified in A.

## Question 9

Correct answer: **D**

Justification:

A is **incorrect** because the story is not constructed correctly. Also, the acceptance criteria should be defined before designing test cases.

B is **incorrect** because the story is not constructed correctly: the story is ambiguous about what type of user account should have this access, which would need to be clarified prior to defining the acceptance criteria.

C is **incorrect** as the Given-When-Then syntax is used when defining BDD tests.

D is **correct** because the story is ambiguous around which user accounts should have the access requested. Testers bring unique perspectives to the review of the story, and are able to assist in

clarifying the user story while it is being written and reviewed with the business owner and developer.

## Question 10

Correct answer: **B**

Justification:

A is **incorrect** because the retrospective does not aim to get user experience of other projects, but to gather experiences from all stakeholders who have an input into team activities on the current Agile project.

B is **correct** because the retrospective requires the attendance of all stakeholders who have an input into team activities.

C is **incorrect** because during the retrospective the team focus is on how to improve the processes and velocity.

D is **incorrect** because the retrospective can conclude an iteration, not a project, and if games are used, they aim to help people to be more transparent.

## Question 11

Correct answer: **D**

Justification:

A is **incorrect** because testers should attend the retrospective regardless of whether they have anything to contribute; the whole team should be present and work towards improvements within the team.

B is **incorrect** because testers play an important part in a retrospective as they are part of the team and bring their unique perspective.

C is **incorrect** because testers (and all team members) should be

encouraged to discuss successes and opportunities for improvement across the whole team, not just discuss testing.

D is **correct** because, as mentioned in C, testers (and all team members) should be encouraged to discuss successes and opportunities for improvement across the whole team, not just discuss testing.

## Question 12

Correct answer: **B**

Justification:

A is **incorrect** because the code is generated by the developer. The CIS builds, integrates and runs tests on the software components.

B is **correct** as the CIS is used to assess the quality of all committed code.

C is **incorrect** because, even though this is possible, it is not the main goal.

D is **incorrect** because the CIS is a shared system used by all the developers.

## Question 13

Correct answer: **B**

Justification:

A is **incorrect** because release planning is high level and tasks are defined during iteration planning.

B is **correct** because testing tasks, amongst others, are defined during iteration planning sessions.

C is **incorrect** because testing and quality are discussed during release and iteration planning.

D is **incorrect** because planning meetings do not execute testing, or

any other, tasks.

## Question 14

Correct answer: **D**

Justification:

- i. is **true** as business stakeholders are involved in testing activities, either formally or via experimentation when features are developed.
- ii. is **false** as test automation is not **mandatory** but it is highly recommended to be able to continually verify features previously delivered and allow testers time to test new features.
- iii. is **false** as system test documentation is still produced on Agile projects, but the detail in the documentation is enough to provide information relevant to the team. Formal detailed test plans and test cases are often replaced with checklists, exploratory session notes or other documentation that satisfies the team's objectives.
- iv. is **true** as with all development teams, quality is the responsibility of everyone on the team, but it is enforced more on Agile teams.

A is **incorrect** because (ii) and (iii) are false and (i) and (iv) are true.

B is **incorrect** because (ii) and (iii) are false and (i) and (iv) are true.

C is **incorrect** because (ii) and (iii) are false and (i) and (iv) are true.

D is **correct** because (ii) and (iii) are false and (i) and (iv) are true.

## Question 15

Correct answer: **C**

Justification:

A is **incorrect** because testers are not expected to write unit tests; this is still a developer task. Testers can contribute their automated system tests to the CIS build, though.

B is **incorrect** because there is significantly more test automation used on Agile projects, but there is no expectation to automate *all* system test cases.

C is **correct** because testers are expected to contribute to the acceptance criteria of user stories.

D is **incorrect** because while some teams encourage testers to contribute to user stories, it is not expected; nor is it expected that testers write the user stories.

## Question 16

Correct answer: **B**

Justification:

A is **incorrect** because testers being embedded in an Agile team face the possibility of losing independence, and developers writing and automating the majority of the tests reduces the level of test independence significantly.

B is **correct** because the tester is part of the Agile team on a long-term basis, therefore building trust with the team to maintain the whole-team approach. The tester provides independent testing to the team and is also part of a larger test team that they can turn to for support and guidance if required. This answer meets both criteria in the question – independence and maintaining the whole-team approach.

C is **incorrect** because while this is a type of test independence on an Agile team, it does not conform to the ‘maintaining the whole-team approach’. Having testers come in at the end of the sprint can cause trust issues within the team, as the tester has not been part of

the team from the beginning.

D is **incorrect** because developers would still be testing their own code if the tester was only providing consultancy services, therefore losing a level of test independency.

## Question 17

Correct answer: **C**

Justification:

A is **incorrect** because performance and security testing are specialist skillsets. Jane may have the skills to perform this testing, but it may need to run as a parallel activity to other tasks within that iteration.

B is **incorrect** because, to maintain independence, performance and security testing should be done by specialists in these fields.

C is **correct** because Jane is also part of the company's test team and she should first seek assistance from within the company for these specialist skills.

D is **incorrect** because Jane should first check within the company's test team for these skills before seeking external consultancy.

## Question 18

Correct answer: **C**

Justification:

A is **incorrect** because the test management tool would not provide the product owner with an overall view of the project testing status as not all tests may be captured in the test management tool at the time.

B is **incorrect** because defect status is one part of assessing product

quality, but does not provide an overview of the project testing status.

C is **correct** because the story board or task board provides a full view of all user stories and tasks the team have committed to for the iteration, and includes test status. The board would provide the product owner with a clear indication of the project testing status against delivered functionality.

D is **incorrect** because the team burndown chart provides a statistical view of the progress of all user stories against estimated effort; it does not provide the granularity needed to provide an overview of the project testing status.

## Question 19

Correct answer: **C**

Justification:

A is **incorrect** because the issues blocking progress of the user stories are not actually defects, but lack of test data, test environment configuration and so forth, and would need more attention than raising a defect.

B is **incorrect** because there is no guarantee that the right people are reading the daily test status report.

C is **correct** because the whole team need to be aware of the issues preventing progress and, as a team, work on a way of unblocking your tasks.

D is **incorrect** because this goes against the whole-team approach by making the iteration manager a pseudoproject manager. Issues blocking progress should be raised with the whole team.

## Question 20

Correct answer: **B**

Justification:

A is **incorrect** because estimation processes on Agile teams ensure that the team only commits to the work that they can achieve in an iteration.

B is **correct** because change is expected in Agile development, therefore there is a much higher rate of code churn than in traditional teams.

C is **incorrect** for the same reasons as A.

D is **incorrect** because not all Agile teams use the latest cutting edge technologies.

## Question 21

Correct answer: **D**

Justification:

A is **incorrect** as automation at all test levels is required to help reduce regression risk in an Agile project.

B is **incorrect** as automation at all test levels is required to help reduce regression risk in an Agile project. It is recommended, based on the test pyramid, that more automation be conducted at lower test levels (unit and integration); however, this question refers to overall test automation, not volume of test automation.

C is **incorrect** for the same reasons as B.

D is **correct** as automation at all test levels is required to help reduce regression risk in an Agile project.

## Question 22

Correct answer: **B**

Justification:

A is **incorrect**. The quality of user stories is shared between all

team members.

B is **correct**. Testers on Agile teams coach other team members about tests and ensure that all tests are executed.

C is **incorrect**. This activity is for the Scrum master.

D is **incorrect**. Unit tests are only a type of test, even in an Agile team, and are generally the responsibility of the developer.

## Question 23

Correct answer: **D**

Justification:

**A** is **incorrect**. The whole team is cross-functional, not each team member, but the tester must understand what the others do.

**B** is **incorrect**. The whole team is responsible for different testing activities.

**C** is **incorrect**. There is a risk that testers may lose a level of independence if they are involved with developers' tasks; however, a tester must work with all team members and should be involved with all team members' activities.

**D** is **correct**. Each team member is self-organised.

## Question 24

Correct answer: **A** and **C**

Justification:

**A** is **correct** because this is a TRUE statement and is one of the tasks of a tester in an Agile team.

**B** is **incorrect** because this is a FALSE statement; as the tester, you are not expected to be involved in code review sessions.

**C** is **correct** because this is a TRUE statement; testers are expected to participate in retrospective meetings and provide feedback on all

team activities, not just on testing.

D is **incorrect** because this is a FALSE statement; testers contribute to defining the acceptance criteria, along with developers and business representatives, and therefore are not ‘responsible’ for defining them.

E is **incorrect** because this is a FALSE statement; while testers contribute to the review and acceptance criteria of a user story, they generally do not assist in soliciting the business requirements (user stories).

## Question 25

Correct answer: **C**

Justification:

A is **incorrect** because the definition of a test scenario is: ‘A document specifying a sequence of actions for the execution of a test; also known as test script or manual test script.’

B is **incorrect** because the definition of session-based testing is: ‘An approach to testing in which test activities are planned as uninterrupted sessions of test design and execution, often used in conjunction with exploratory testing.’

C is **correct** because the definition of a test charter is: ‘A statement of test objectives and possibly test ideas about how to test. Test charters are used in exploratory testing.’

D is **incorrect** because the definition of a test oracle is: ‘A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual or an individual’s specialized knowledge, but should not be the code.’

(These definitions are from the ISTQB® Glossary and have been used with

permission.)

### Question 26

Correct answer: **B**

Justification:

A is **incorrect**. This is the user story format.

B is **correct**. The Given-When-Then is the Gherkin format much used in BDD.

C is **incorrect**. This is an adaptation of the BDD format.

D is **incorrect**. This is the typical if structure in most programming languages.

### Question 27

Correct answer: **C**

Justification:

A is **incorrect** as the test pyramid emphasises having a large number of tests at the lower levels, in which unit testing forms the base of the test pyramid.

B is **incorrect** for the same reasons as A.

C is **correct** as the test pyramid emphasises having a large number of tests at the lower levels, in which unit testing forms the base of the test pyramid.

D is **incorrect** for the same reasons as A.

### Question 28

Correct answer: **B**

Justification:

A is **incorrect** because the test pyramid is used to emphasise a larger number of tests at the lower test levels – for example, unit

test level – in order to prevent defects earlier.

B is **correct** because the testing quadrants are used to align the test levels with the appropriate testing types, which helps to ensure all important test levels and test types are included.

C is **incorrect** because test-driven development is a way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

D is **incorrect** because behaviour-driven development allows a developer to focus on testing the code based on the expected behaviour of the software.

## Question 29

Correct answer: **C**

Justification:

A is **incorrect**. This is not where user acceptance testing (UAT) is shown in the testing quadrants. This quadrant focuses on low-level tests, such as unit and component tests, and therefore would not include any user acceptance testing.

B is **incorrect**. This is not where UAT is shown in the testing quadrants. This quadrant would have acceptance test level tests that are used to verify that the acceptance criteria of a user story are met, but it is not UAT, i.e. ATDD (acceptance test-driven development) tests.

C is **correct**. UAT is included in the examples of the testing quadrant Q3. This is not to be confused with tests in Q2, which are often developed through the ATDD technique.

D is **incorrect**. This is not where UAT is shown in the testing quadrants. This quadrant focuses more on technical testing, such as security and performance testing, and therefore would not include any user acceptance testing.

## Question 30

Correct answer: **D**

Justification:

A is **incorrect** as it assumes you plan everything before you start executing tests. This is actually not an impossible scenario, but D is what you should strive for.

B is **incorrect** as it assumes you would plan everything before execution, and only test in the next sprint. This is actually not an impossible scenario, but D is what you should strive for.

C is **incorrect** as it assumes you would do unit tests only. This is actually not an impossible scenario, but D is what you should strive for.

D is **correct**. It is always a good idea to use incremental test design to get to test earlier and to keep up to speed with the Agile project.

## Question 31

Correct answer: **A and D**

Justification:

A is **correct** as Story B has a higher risk and impact assessment and therefore requires more testing rigour than Story A and Story C. Story C has a higher impact assessment than Story A and therefore may require slightly more testing rigour than Story A. Answer A meets these conditions.

B is **incorrect** as Story B has a higher risk and impact assessment, therefore requires more testing rigour than Story A and Story C. Story C has a higher impact assessment than Story A, therefore may require slightly more testing rigour than Story A. Answer B does not meet these conditions.

C is **incorrect** as Story B has a higher risk and impact assessment,

therefore requires more testing rigour than Story A and Story C. Story C has a higher impact assessment than Story A, therefore may require slightly more testing rigour than Story A. Answer C does not meet these conditions.

D is **correct** as Story B has a higher risk and impact assessment, therefore requires more testing rigour than Story A and Story C. Story C has a higher impact assessment than Story A, therefore may require slightly more testing rigour than Story A. Answer D meets these conditions.

E is **incorrect** as Story B has a higher risk and impact assessment, therefore requires more testing rigour than Story A and Story C. Story C has a higher impact assessment than Story A, therefore may require slightly more testing rigour than Story A. Answer E does not meet these conditions.

### Question 32

Correct answer: **B**

Justification:

A is **incorrect** as consensus has not been reached for the first estimation and another iteration is necessary, when the estimators of the lowest and highest score will have to defend their estimations.

B is **correct** as consensus has not been reached for the first estimation and another iteration is necessary, when the estimators of the lowest and highest score will have to defend their estimations.

C is **incorrect** for the same reasons as A.

D is **incorrect** for the same reasons as A.

### Question 33

Correct answer: **B**

Justification:

A is **incorrect** because discussion on how the system performs during peak operations will be helpful for load and performance testing, but this is not the BEST option to support functional testing.

B is **correct** because defect analysis of previously implemented systems can show areas of concern that may require additional attention during your testing, especially functional areas that were previously prone to defects.

C is **incorrect** because the project plan and project schedule is too high level for detailed testing activities.

D is **incorrect** because project risks are related to the overall project (resources, timeframes etc.); product risks would have been more relevant to testing activities.

### Question 34

Correct answer: **A**

Justification:

- i. is **correct** as this is a testable acceptance criterion as it defines values for functional verification of system behaviour.
- ii. is **correct** as this is a testable acceptance criterion as it defines a valid negative functional path of system behaviour.
- iii. is **incorrect** as this is NOT a testable acceptance criterion, ‘easily’ is an ambiguous word and needs to be quantified before it can be considered a testable acceptance criterion.
- iv. is **correct** as this is a testable acceptance criterion as it defines access required for verification of system behaviour.
- v. is **incorrect** as this is NOT a testable acceptance; ‘quickly’ is an

ambiguous value and needs to be quantified before it can be considered a testable acceptance criterion.

A is **correct** because (i), (ii) and (iv) are true and (iii) and (v) are false

B is **incorrect** because (i), (ii) and (iv) are true and (iii) and (v) are false

C is **incorrect** because (i), (ii) and (iv) are true and (iii) and (v) are false

D is **incorrect** because (i), (ii) and (iv) are true and (iii) and (v) are false

### Question 35

Correct answer: **B**

Justification:

A is **incorrect** as the acceptance criterion relates to performance metrics and functional behaviour is defined as: ‘The externally observable behaviour with user actions as input operating under certain configurations.’

B is **correct** as the acceptance criterion relates to performance metrics and quality characteristics are defined as: ‘How the system performs the specified behaviour. The characteristics may also be referred to as quality attributes or non-functional requirements. Common quality characteristics are performance, reliability, usability, etc.’

C is **incorrect** as the acceptance criterion relates to performance metrics and business rules are defined as: ‘Activities that can only be performed in the system under certain conditions defined by outside procedures and constraints (for example, the procedures used by an insurance company to handle insurance claims).’

D is **incorrect** as the acceptance criterion relates to performance

metrics and constraints are defined as: ‘Any design and implementation constraint that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight, and interface connections.’

(These definitions are from the ISTQB® Glossary and have been used with permission.)

### Question 36

Correct answer: **C**

Justification:

A is **incorrect** because this would be more a unit level or test-driven development test case, verifying low-level system functionality.

B is **incorrect** because this test is written in a behaviour-driven development test case format.

C is **correct** because this is the BEST example of an acceptance test-driven development test case; it is verifying the functionality at an acceptance test level by use of example.

D is **incorrect** because this test case does not relate to the functionality defined in the user story. The syllabus states: ‘The examples must cover all the characteristics of the user story and should not add to the story.’ It would be expected that another user story would cover the maintenance of credit card details.

### Question 37

Correct answer: **B**

Justification:

A is **incorrect** as boundary value analysis test design techniques

would not provide the BEST test coverage for this user story in comparison to state transition testing techniques.

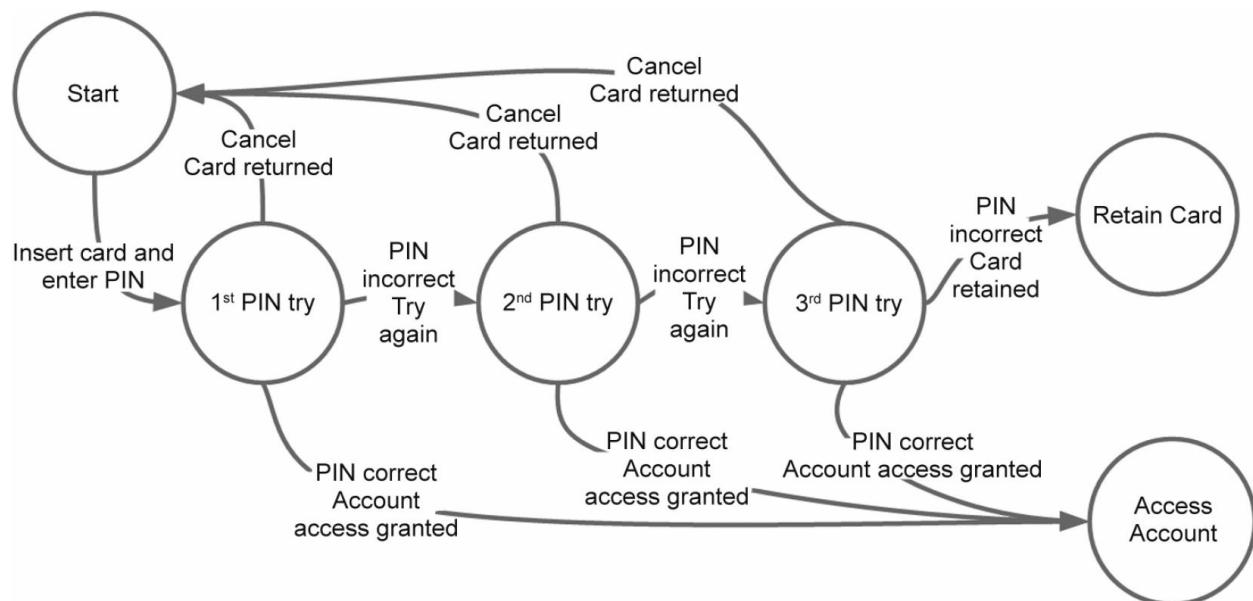
B is **correct** as the focus in this user story is the state transitions of an ATM, therefore state transition testing is the best option, as shown in [Figure 5.1](#).

C is **incorrect** as decision tables test design techniques would not provide the BEST test coverage for this user story in comparison to state transition testing techniques.

D is **incorrect** as use case test design techniques would not provide the BEST test coverage for this user story in comparison to state transition testing techniques.

---

**Figure 5.1 State transitions for ATM PIN user story**



### Question 38

Correct answer: **C**

Justification:

A is **incorrect** because the purpose of the session is not to survey

and explore the web browser configurations, but to survey, explore and test the application under test using the various web browser and configuration settings.

B is **incorrect** because the charter activities should be defined before commencing the exploratory test session.

C is **correct** because exploratory testing is about doing parallel test design and test execution against the charter activities, and recording the results.

D is **incorrect** because exploratory testing is not an ad hoc testing exercise.

### Question 39

Correct answer: **A**

Justification:

A is **correct** as these are main functions of the continuous integration process that is automatically run by continuous integration tools.

B is **incorrect** as automated test execution tools do not build or deploy applications.

C is **incorrect** as data loading tools do not build, test or deploy applications, or perform static verification.

D is **incorrect** as exploratory testing tools do not build or deploy applications, or perform static verification.

### Question 40

Correct answer: **A and C**

Justification:

A is **correct** as mind maps can be very effective to quickly design and define tests for a new feature.

B is **incorrect** as running automated tests requires specific tools for this type of activity.

C is **correct** as exploratory tests involve parallel test design and execution, therefore mind maps can also be very effective here.

D is **incorrect** as implementation and running load tests require specific tools.

E is **incorrect** as measuring code coverage requires specific unit level code metrics tools.

# INDEX

- 3C (card, conversation, confirmation) [25–6, 128–9](#)
- acceptance criteria [xvii, 129–30](#)
- acceptance test-driven development (ATDD) [79, 81–3, 139–42](#)
- acceptance tests [9, 15, 48](#)
- Agile approaches
  - Extreme Programming (XP) [15–18](#)
  - Kanban [21–2](#)
  - Scrum [18–21](#)
- Agile Manifesto [xvii, 3–6](#)
- Agile project techniques *see* techniques
- Agile project tools *see* tools
- Agile software development
  - overview [xvii, 1–2](#)
  - Agile Manifesto [3–6](#)
  - approaches [15–22](#)
  - change, responding to [32–3](#)
  - continuous integration [29–30](#)
  - feedback, early and frequent [10–11](#)
  - iteration planning [32](#)
  - release planning [30–2](#)
  - retrospectives [27–9, 73–4](#)
  - testing and process [47](#)
  - user story creation [22–7](#)
  - whole-team approach [7–9](#)
- Agile testing methods *see* methods
- Agile testing vs traditional testing

overview 37

configuration management and 50

development activities and 38–41

organisational options for independent testing 51–2

project work products 41–5

test levels 45–9

ATDD *see* acceptance test-driven development (ATDD)

automated test cases 62–5

automated test execution tools 166–8

automation xviii, 40–1, 44, 63, 65

backlogs

- iteration 156–7
- product 19–20, 23, 73, 157
- release 32–3
- sprint 20

baskets 21

BDD *see* behaviour-driven development (BDD)

behaviour

- interpersonal 70–1
- modelling 137–8
- non-functional behaviour 127

behaviour-driven development (BDD) 79, 83–4

black-box test design 134–8

blended test strategies 143–4

boundary value analysis 135–7

build verification test (BVT) xvii

burndown/burnup chart 21

business skills 68

Business-Facing tests 87

business-oriented work products 41–2

card, conversation and confirmation (3C) 25–6, 128–9

catastrophic failure 107

change

- documentation and 41

response to 4, 5, 32–3, 40

CI *see* continuous integration (CI)

class, responsibilities and collaboration (CRC) cards 15, 16

cloud computing tools 170–1

coaching

- in Scrum 19
- testers' skills 70–1

coding 15

collaboration

- CRC cards 15, 16
- with customer 4, 5, 9–10
- to meet goals 89
- user stories and 8, 22–7

‘collective code ownership’ 64

co-location 8, 17, 89

combinatorial explosion 104–6

commitment 89–90

communication

- with team and customer 6, 8–9, 81
- of test status 56–62
- tools 158–60
- as value 3, 16

complexity 109

configuration item xvii

configuration management

- overview xvii, 50
- test cases and 62
- tools 161–2

confirmation 26

continuous integration (CI)

- BDD, ATDD and 84
- configuration management 50
- process 18, 29–30
- quality and 10–11
- testing and 93–4

continuous testing tools 169–70  
conversations 25–6  
courage 16–17  
CRC (class, responsibilities and collaboration) 15, 16  
create, read, update, delete (CRUD) 146  
credibility 89  
Crispin, Lisa 84–7  
critical failure 107  
Critiquing the Product 86–7  
cross-functional teams 8, 89–90  
cross-team working 9–10  
customer satisfaction 10–11

daily Scrum 20  
defects 109–10  
definition of ‘done’ (DoD)  
for Agile projects 40, 147–8  
measurement of 132–4  
in Scrum 20  
user stories and 128–9

development  
Agile vs traditional approaches 38–41  
tools 43  
work products 42–3

diversity 17

documentation  
of Agile tests 138–9  
‘just enough’ 3–4  
project work products 41–5

DoD *see* definition of ‘done’ (DoD)

dynamic analysis 30  
dynamic test strategies 143–4

economics 17  
empowerment 6, 89  
enterprise culture 90

entry and exit criteria 47, 49  
epics 42  
equivalence partitioning 135–7  
estimation of testing effort 119–22, 157  
exploratory testing  
    in Agile projects 44, 144–6  
    definition xvii  
    test pyramid and 85  
    tools 168–9  
Extreme Programming (XP) 15–18

face-to-face conversation 6  
failure 17, 64, 104–9  
features  
    requests 10  
    verification, validation and delivery 47–8  
feedback 10–11, 16, 90  
functional testing skills 68

‘Given-When-Then’ tests 26–7, 83–4  
Gregory, Janet 84–7  
Guiding Development 86–7

handovers 7–8  
heuristic test strategies 143–4

improvements 17, 27–9  
incremental development model xvii  
incremental test design 17, 18, 97–8  
information  
    gaps 131–2  
    sources of 130–2  
    tools for sharing 158–60  
interactions 3, 9–10  
interfaces 130–1  
interpersonal skills 70–1

INVEST acronym 24–5  
iteration backlog 156–7  
iteration planning 32, 73  
iterative development model xviii  
iterative lightweight quality risk analysis 110–13

‘just enough’ documentation 3–4  
just-in-time (JIT) flows 21

Kanban 21–2

likelihood 109–12  
logging test results 146–7

manual test cases 62–5  
marginal failure 108  
MBT (model-based testing)  
    automated test design 118  
    tools 163–4

meetings  
    iteration planning 31–2  
    release planning 31  
    retrospectives 27–9, 73–4  
    stand-up 8, 20

methods  
    overview 78  
    role of the tester 88–98  
    test pyramid 84–6  
    test-first methods 78–84  
    testing quadrants 86–8

metrics 61–2  
mind mapping 98, 162–3  
minimum viable product (MVP) 10  
misunderstandings 10  
model-based testing (MBT)  
    automated test design 118

tools 163–4

momentum 11

mood 60–1

multiple test methods 106

mutual benefit 17

MVP (minimum viable product) 10

negligible failure 108

Niko-niko calendar 60–1

non-functional behaviour 127

opportunity 17

pairing

- pair programming 15, 18
- testers and 39, 95–7

PBI *see* product backlog items (PBIs)

performance testing xviii

planning 16, 93–4

planning poker 18, 119–22

‘Power of Three’ 10

probability 109–12

product backlog 19–20, 23, 73, 157

product backlog items (PBIs) 19–20 *see also* user stories

product increment 19

product risk xviii

productivity 11

progress 6, 11

project management 16

project momentum 11

project work products

- Agile vs traditional approaches 41–5
- tools 161–2

pull principle 156

pyramids 84–6

quality

- characteristics 24, 68
- as objective of XP 17
- resolving problems early 10–11
- responsibility for 7–8, 39
- testers as coach 70–1
- user stories 24–5

quality risk

- analysis 103, 110–18
- definition xviii

reactive testing strategies 143–4

redundancy 17

reflection 6, 17

regression risk 62–5

regression testing xviii, 46–7, 64

release backlog 32–3

release planning 30–2

requirements 5, 10, 22, 73

resilience 90

responsibility 7–8, 17

retrospectives 27–9, 73–4

risk 23, 30, 74, 104

risk charts 116–18

risk poker 113–16, 121

risk priority 112–13

risk-based testing 102–10

satisfaction surveys 60

Scrum

- as Agile approach 18–21
- incremental test design 97–8
- integration in 93–4
- mind mapping 98
- model for software development 46–7
- pairing 95–7

Sprint Zero 91–3  
teams 19, 88–91  
test planning 94–5  
self-organising teams 6, 19, 89  
self-reflection 6, 17  
session-based test management 168–9  
simplicity 6, 16  
sitting together 8, 17, 89  
skills 9–10, 67–72, 131  
slack 18  
software  
    build and distribution tools 160–1  
    design as holistic process 142–3  
    lifecycle xviii  
    working 3–4  
software development  
    Extreme Programming (XP) 15–18  
    Kanban 21–2  
    Scrum 18–21  
    sprints 19–20  
    stand-up meetings 8, 20  
    status 56–62  
    story cards 42  
    Supporting the Team 86–7  
    surveys 60  
    sustainable development 6  
    system wide integration 15  
  
task boards 57–8, 93–4, 156–7  
task management and tracking tools 156–8  
TDD *see* test-driven development (TDD)  
teams  
    communication in 3, 8–9, 81  
    cross-functional 8, 89–90  
    self-organising 6, 19, 89  
    testers within 51–2, 72–4, 88–91

whole-team approach 7–9, 17  
working in 88–91

technical debt 40

technical testing skills 68–70

techniques

- acceptance test-driven development (ATDD) 139–42
- black-box test design 134–8
- definition of ‘done’ 132–4, 147–8
- documenting Agile tests 138–9
- exploratory testing 144–6
- information, sources of 130–2
- logging test results 146–7
- reactive testing strategies 143–4
- software design to pass tests 142–3
- test bases, determining 126–8
- testable acceptance criteria 129–30
- user stories and DoD 128–9

Technology-Facing tests 87

ten-minute build 18

test automation xviii, 40–1, 44, 63

test bases xviii, 41, 126–8

test cases

- management tools 164–5
- managing regression risk 62–5

test charters xviii, 144–5

test data load tools 165–6

test data preparation and generation tools 165

test development 130–2

test oracle xix

test pyramid 84–6

test work products 43–5

testable acceptance criteria 129–30

test-driven development (TDD) xviii, 43, 78–81

testers

- in Agile process 38–9, 47

incremental test design and 97–8  
integration and 93–4  
integration in team 51–2, 72–4  
mind mapping and 98  
pairing and 95–7  
skills of 67–72  
Sprint Zero and 91–3  
teamwork and 88–91  
test planning and 94–5

test-first  
methods 78–84  
programming 18

testing  
acceptance testing 15  
Agile vs traditional 38–41  
continual testing 38–9  
design 134–8, 162–4  
estimating effort 119–22  
execution of xviii, 166–8  
independence of 51–2  
levels of 45–9, 84–5  
logging results 146–7  
method selection 117–18  
planning 43–4, 94–5  
status of 56–62  
strategies xix, 43–4, 72–3, 143–4  
unit testing 15

testing effort 119–22  
testing quadrants 86–8  
time boxing 20, 145  
tools

overview 154–5  
automated test execution 166–8  
cloud computing 170–1  
communication and information sharing 158–60

configuration management 161–2  
continuous testing 169–70  
exploratory test 168–9  
software build and distribution 160–1  
task management and tracking 155–8  
test case management 164–5  
test data load 165–6  
test data preparation and generation 165  
test design 162–4  
test execution 166–8  
virtualisation 170–1  
traceability 113  
transparency 20, 89

unidimensional test basis 127, 128  
unit test frameworks *xix*, 166–8  
unit testing 15, 18, 79–81  
user acceptance tests 9, 48  
user stories  
    collaboration and 22–7  
    definition *xix*  
    definition of ‘done’ (DoD) and 128–9  
    documentation 41–2  
    risk visualisation 116–17

validation 47, 117–18, 146  
velocity 19  
verification 47  
videoconferencing 160  
virtualisation tools 170–1  
visualisation 116–17, 157–8  
V-model 46

waterfall model 45  
white-box tests 87  
whole-team approach 7–9, 17

wikis [158–9](#)

working hours [17](#)

working software [3–4, 5](#)

work-in-progress limit [21](#)

workspaces [8, 17, 18–19](#)

X-driven development [78–9](#)

XP (Extreme Programming) [15–18](#)

xUnit frameworks [166–8](#)

# AGILE TESTING FOUNDATIONS

## An ISTQB Foundation Level Agile Tester guide

Rex Black (editor)

Agile is an iterative approach to software development that has rapidly gained popularity in the wider IT industry. For software testers, Agile testing brings many advantages to teams, from increasing overall product quality to providing greater scope for flexibility.

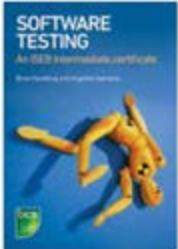
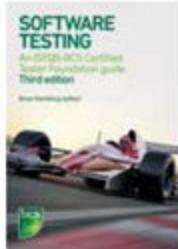
Building on the ISTQB Foundation Level Agile Tester syllabus, this book covers Agile principles, methods, techniques and tools in the context of software testing. The book is perfect for software testers interested in the benefits of Agile testing, working in an Agile environment or undertaking the ISTQB Foundation Level Agile Tester exam.

- Understand Agile and its principles
- Explore the benefits of applying Agile to software testing
- Discover Agile tools to use in your testing projects
- Test your knowledge with questions at the end of every chapter and a complete Agile Tester sample exam

### ABOUT THE AUTHOR

Rex Black is president of RBCS Inc. and has authored over a dozen training courses. He leads the Agile Working Group, is the coordinator for the Advanced Test Manager syllabus team, and was previously president of the ISTQB and ASTQB. Rex is also a prolific author with over 10 books on software testing under his belt.

You might also be interested in:



*A book that I am confident will become a milestone in the testing domain and a reference for the Agile community.*

*Gualtiero Bazzana,  
ISTQB® President*

Information Technology

Cover photo: Shutterstock © Christopher Waters

ISBN 978-1-78017-336-8



9 781780 173368

