Chapter

1 Introduction

Hmm, The Wheel you say! Well, I don't wish to belittle your achievement, but I've traveled far and wide and I've seen a great many of these things invented by a great many people in a great many different caves!

Big Ugg, Neander Valley, 35,000 B.C.

1.1 Purpose of the Book

This book provides comprehensive and rigorous guidance to workers in the field of software testing for researching or setting up a software testing process within organizations.

The book provides advice and guidance on all aspects of the testing process, including:

- The need to test software and the approach to testing
- Specific details of testing techniques with worked examples
- The planning and management of testing projects
- Testing roles and responsibilities
- Comprehensive details of the testing phases
- Extensive testing document templates, proformas, and checklists
- Recommendations for testing process improvement and the role and use of metrics
- The testing issues facing developers of Object-Oriented and Component-Based systems.

The book covers the testing of software from a number of sources, including software developed or modified in-house, software that represents the modification or extension of exisiting legacy software systems, and software developed on behalf of an organization by a third party.

The book also covers the acceptance testing of *commercial off-the-shelf (COTS)* software procured by an organization, or COTS software that has undergone development either internally or by a third party on behalf of an organization.

This book should be used in a pragmatic manner, in effect providing a testing framework that can be used by all members of staff involved in software development and testing within an organization to improve the quality of the software they deliver and to reduce timescales, effort, and cost of testing.

Alternatively, the testing process described in this book can be customized to match the specific testing requirements of any particular organization, and a series of real-world case studies are provided to illustrate how this can be achieved.

1.2 Readership

The target audience for this book includes the following people:

- **Technical Director/Managers** who need to improve the software testing process within their organization (in terms of quality, productivity, cost, and/or repeatability of the process)
- **Quality Assurance (QA) professionals** (such as company QA Directors or Managers) who need to put in place a formal organization-wide approach to software testing
- **Project Managers/Leaders** who need to save time, effort, and money and improve quality by adopting a complete, standard, off-the-shelf solution to their testing requirements
- **Independent Information Technology (IT), QA,** or **Management Consultants** who provide advice and guidance to clients on their software testing process, for whom the book will represent a key item in their "Consultants Tool Kit"
- **Testing/QA Professionals** (such as Test Analysts, Testers, or QA Representatives) who wish to save time and effort by adopting predefined testing artifacts (such as standard templates for Test Script, Test Plan, and Test Specification documents)
- **IT Professionals** who need to understand the software testing process (such as developers involved in Unit or Integration testing)
- **Any staff members** who are keen to improve their career prospects by advocating a complete testing solution to their organizations' software testing needs, particularly where there is a need to improve quality or save time, effort, and cost
- **Training Managers/Trainers** who are in the process of writing or amending testing training materials and who need to obtain a pragmatic view of the testing process and its application
- **Students** who need to obtain a pragmatic/real-world view of the application of testing theory and principles to organizational software testing requirements, or who have an interest in testing-process improvement and the role and use of metrics.

1.3 How to Read This Book

This book is divided into three parts, all closely linked, but each of which can be read and applied separately.

Part 1 (Chapters 2–13) documents the "traditional view" of the components comprising a software testing process. Part 1 provides detailed information that can be used as the basis for setting up a testing-process framework tailored to the individual requirements of any organization involved in software testing.

Part 2 (Chapters 14–18) provides a series of case studies that show how a number of organizations have implemented their own testing process based on the "classic view" described in Part 1. These case studies can be read to provide real world

guidance on how an individual organization can implement a testing-process framework to meet its own testing requirements.

Part 3 (the appendices) contains a set of standard testing document templates, proformas, and checklists plus a number of appendices that expand on topics described in passing in the main body of the book. The standard testing document templates, proformas, and checklists are also available from the following URL: <us.cambridge.org/titles/052179546X> so that they can be used immediately without modification or customized to reflect the particular requirements of any organization (such as a corporate style, branding, or documentation standard).

Terms in italics are fully defined in the glossary.

1.4 Structure and Content of This Book

Specifically, the chapters and appendices comprising this book are:

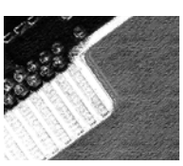
- Chapter 2, which discusses just how challenging it is to thoroughly test even the most simple software system, reviews a number of definitions of testing, provides a brief overview of the approach to software testing, and lists definitive testing references for further reading.
- Chapter 3, which describes the principal techniques used in designing effective and efficient tests for testing software systems and provides, where appropriate, references to illustrative worked examples in the appendices.
- Chapter 4, which deals with the issues associated with the management and planning of the testing process, provides guidance on the organization of testing and testing projects and on the need for thorough planning, describing a number of techniques for supporting the planning process.
- Chapters 5–11, which provide details on each of the testing phases (from Unit Testing to Acceptance Testing and on to Regression Testing[1]) and their interrelationships. Each chapter is presented in a standard format and covers:
  - *the overall testing approach for that phase*
  - *test data requirements for that phase*
  - *the roles and responsibilities associated with that phase*
  - *any particular planning and resourcing issues for that phase*
  - *the inputs to and the outputs from that phase*
  - *a review of the specific testing techniques that are appropriate to that phase.*
- Chapter 12 considers the need for process improvement within the testing process and reviews the role of metrics (proposing a pragmatic metrics set that can be used effectively within and across testing projects). It also provides references to further sources of information on test process improvement.

[1] While not strictly speaking a separate testing phase, Regression Testing is included in this list for the sake of completeness.

- Chapter 13, which for organizations adopting the testing process described in this book or using it as the basis for setting up their own testing-process framework, discusses the approach to introducing testing into an organization and managing its successful adoption, reviews the need to maintain that testing process, and proposes an approach to satisfy this requirement.
- Chapters 14–18 provide a series of real-world case studies describing how a number of commercial organizations have implemented their own customized view of the testing process described in Chapters 2–13. Specifically, the organizations covered in the case studies are:
  - *The British Library*
  - *Reuters Product Acceptance Group*
  - *Crown Quality Assurance Group*
  - *The Wine Society*
  - *Automatic Data Processing (ADP) Limited.*
- Appendices A–J provide a set of testing document templates, proformas, and checklists:
  - *terms of reference for testing staff*
  - *summary testing guides for each testing phase*
  - *a Test Plan document template*
  - *a Test Specification document template*
  - *a Test Script template*
  - *a Test Result Record Form template*
  - *a Test Log template*
  - *a Test Certificate template*
  - *a Re-use Pack checklist*
  - *a Test Summary Report template.*
- Appendices K–M present a series of worked examples of testing techniques described in Chapter 3.
- Appendices N–Q expand on topics described in passing in the main body of the book and include:
  - *a scheme and set of criteria for evaluating the relative merits of commercially available automated software testing tools*
  - *an overview of the process of Usability Testing and its application*
  - *a scheme and set of criteria for performing an audit of a testing process*
  - *a discussion of the issues involved in the testing of object-oriented and component-based applications.*
- A list of the references cited in the book.
- A glossary of terms used in this book.

Part 1

The Traditional Testing Process



Chapter

2 An Overview of Testing

As we strive to implement the new features of our applications, there is one thing we can say with absolute certainty – that at the same time, we also introduce new defects.

2.1 Introduction

This chapter gives an overview of testing in order to provide an understanding of what testing is and why it is such a challenge, and to emphasize that whenever we test software, the process must be made to be as efficient and effective as possible.

Readers familiar with the need for efficient and effective testing may not find it necessary to read this chapter.

2.2 The Challenge of Testing

So, just how difficult is testing? To help answer this question, consider the following example:

Imagine we have a requirement to test a simple function, which adds two thirty-two–bit numbers and returns the result. If we assume we can execute 1000 test cases per second, just how long will it take to thoroughly test this function?

If you guessed seconds, you are way out. If you guessed minutes, you are still cold. If you guessed hours or days or even weeks, you are not even slightly warm. The actual figure is 585 million years.[1]

But surely, this is a daft example? Nobody in his or her right mind would test such a function by trying out every single possible value! In practice, we would use some formal test design techniques such as *boundary analysis* and *equivalence partitioning* to help us select specimen data to use in our test cases (see Chapter 3 for details

[1] The calculation is quite straightforward (with a calculator): $2^{(32+32)}/1000/60/60/24/365.25 = 584542046$ years.

of test design techniques). Using this test data, we would make the assumption that if the function performed satisfactorily for these specimen values, it will perform satisfactorily for all similar values, reducing the time needed to test the function to an acceptable timescale.

However, as testers, we should not start feeling too confident too soon – there are many other issues that can complicate the testing of our "simple" function. For example:

- What if the function needs to interoperate with other functions within the same application?
- What if the data for the calculation are obtained across a complex Client/Server system and/or the result is returned across the Client/Server system?
- What if the calculation is driven via a complex Graphical User Interface with the user being able to type the addition values into fields and push the buttons to perform the calculation in any arbitrary order?
- What if this function has to be delivered on a number of different operating systems, each with slightly different features, and what if individual users are able to customize important operating system features?
- What if this function has to be delivered on a number of different hardware platforms, each of which could have different configurations?
- What if the application this function belongs in has to interoperate with other applications, and what if the user could be running an arbitrary number of other applications simultaneously (such as e-mail or diary software)?

These are all typical requirements for software systems that a great many testers face every day during their testing careers, and which act to make software systems highly complex and make testing an immense challenge!

2.3 What Is Testing?

The process of testing is by no means new. The Oxford English Dictionary tells us that the term "test" is derived from the Latin expression *testum,* an earthenware pot used by the Romans and their contemporaries in the process of evaluating the quality of materials such as precious metal ores.

Computer programs have undergone testing for almost as long as software has been developed. In the early days of software development there was little formal testing, and debugging was seen as an essential step in the process of developing software.

As the software development process has matured, with the inception and use of formal methods (such as Reference 6), the approach to testing has also matured, with formal testing methods and techniques (such as Reference 8) being adopted by testing professionals.

Most workers in the field of modern software development have an intuitive view of testing and its purpose. The most common suggestions include:

- To ensure a program corresponds to its specification
- To uncover defects in the software
- To make sure the software doesn't do what it is not supposed to do
- To have confidence that the system performs adequately
- To understand just how far we can push the system before it fails
- To understand the risk involved in releasing a system to its users.

Here are some more formal definitions of testing:

> **Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.** (Reference 1)

This definition addresses the traditional testing approach – that is, does the system conform to its stated requirements? This appears to be an intuitive view of testing: we have some statements about how the system should behave, and we confirm that these requirements are met. This approach is also known as *Positive Testing*.

Here is another view of testing:

> **Testing is the process of executing a program or system with the intent of finding defects.** (Reference 2)

This definition is less intuitive and does not, strictly speaking, consider the requirements of the system.[2] Instead, it introduces the notion of actively looking for defects outside the scope of the software requirements, which in practice could be any problem or defect in the system. This approach is also known as *Negative Testing*.

In practice, testing will combine elements of both Positive and Negative testing – checking that a system meets its requirements, but also trying to find errors that may compromise the successful operation or usefulness of the system.[3]

Most recently, the notion of defining testing in terms of risk has become increasingly popular. In this use, the term "risk" relates to the possibility that the *Application Under Test* (*AUT*) will fail to be reliable or robust and may cause commercially damaging problems for the users. Here is a definition of testing in terms of risk:

> **Testing is the process by which we explore and understand the status of the benefits and the risk associated with release of a software system.** (Reference 28)

Within this definition of testing, the role of the tester is to manage or mitigate the risk of failure of the system and the undesirable effects this may have on the user.

Defining testing in terms of risk provides the tester with an additional strategy for approaching the testing of the system. Using a risk-based approach, the tester is involved in the analysis of the software to identify areas of high risk that need to be tested thoroughly to ensure the threat is not realized during operation

[2] Although, a "defect" could be considered to be a failure of the system to support a particular requirement.

[3] It is possible to argue that in a perfect world of complete requirements and accurate specifications, there would be no need for negative testing, since every aspect of the Application Under Test (AUT) would be specified. Unfortunately, the reality is somewhat short of perfection, and so testing is always likely to include a degree of Negative Testing.

of the system by the user. Furthermore, the notion of risk in a project management context is well known and understood, and a great many tools and techniques exist that can be applied to the testing process (such as References 28, 29, and 30).

It may be difficult for the staff involved in the planning and design of tests to identify specific risks for a particular AUT (especially when they may not be familiar with the domain of operation of the software). In assessing risk, it is essential that the following issues be considered:

- The business, safety, or security criticality of the AUT
- The commercial/public visibility of the AUT
- Experience of testing similar or related systems
- Experience of testing earlier versions of the same AUT
- The views of the users of the AUT
- The views of the analysts, designers, and implementers of the AUT.

The need to analyze risk in the testing process is addressed in Chapter 4 – The Management and Planning of Testing.

2.4 Verification and Validation

Another two testing terms, which are frequently used but often confused, are *verification* and *validation*. Reference 40 provides a formal definition of these terms:

> **Verification is the process by which it is confirmed by means of examination and provision of objective evidence that specific requirements have been fulfilled (during the development of the AUT).**
>
> **Validation is the process by which it is confirmed that the particular requirements for a specific intended use (of the AUT) are fulfilled.**

Reference 26 provides a succinct and more easily remembered definition of these terms:

> **Verification: Are we building the product right?**
>
> **Validation: Are we building the right product?**

In essence, verification deals with demonstrating that good practice has been employed in the development of the AUT by, for example, following a formal development process (such as Reference 8).

Validation deals with demonstrating that the AUT meets its formal requirements, and in that respect conforms closely to the Hetzel definition of testing discussed earlier in this chapter (Reference 1).

Both verification and validation (also termed *V&V*) are key to ensuring the quality of the AUT and must be practiced in conjunction with a rigorous approach to requirements management. Chapter 4 provides guidance on the role of requirements management and its role within V&V.

2.5 What Is the Cost of Not Testing?

Many examples exist, particularly where systems have had safety critical, business critical, or security critical applications, where the failure of the system has, either through litigation or loss of public confidence, resulted in the provider of the software going out of business.

Even where a system does not deal with a critical application, failure of high-profile systems, such as an organization's Web Site, free shareware, or demonstration software, can still have serious commercial implications for the organization in terms of loss of public confidence and prestige.

Some defects are very subtle and can be difficult to detect, but they may still have a significant effect on an organization's business. For example, if a system fails and is unavailable for a day before it can be recovered, then the organization may lose a day's effort per person affected. If an undetected defect simply causes the performance of a system to degrade, then users may not even notice that a problem exists. If, however, the defect causes a loss of productivity of just 30 minutes per day, then the organization could lose in the order of 20 days effort per person per year!

2.6 Testing – The Bottom Line

Phrases like "Zero Defect Software" or "Defect Free Systems" are hyperbole, and at best can be viewed only as desirable but unattainable goals.[4]

In practice, it is impossible to ensure that even relatively simple programs are free of defects because of the complexity of computer systems and the fallibility of the development process and of the humans involved in this process.

In simple terms, it is impossible to perform sufficient testing to be completely certain a given system is defect free. When this problem is combined with the fact that testing resources are finite and (more typically) in short supply, then adequate testing becomes problematical. Testers must focus on making the testing process as efficient and as effective as possible in order to find and correct as many defects as possible.

Ultimately, testing can only give a measure of confidence that a given software system is acceptable for its intended purpose. This level of confidence must be balanced against the role the system is intended for (such as safety critical, business critical, secure, confidential, or high-profile applications) and against the risk of the system failing in operation, before the decision to release or to accept software can be made.

[4] Even with mathematically rigorous methods (such as Z and VDM), it is still impossible to say that any but the simplest pieces of software will be defect free.