ACM PRESS BOOKS

This book is published as part of ACM Press Books - a collaboration between the Association for Computing (ACM) and Addison Wesley Longman Limited. ACM is the oldest and largest educational and scientific society in the information technology field. Through its high-quality publications and services, ACM is a major force in advancing the skills and knowledge of IT professionals throughout the world. For further information about ACM, contact:

ACM Member Services
1515 Broadway, 17th Floor
New York, NY 10036-5701
Phone: 1-212-626-0500
Fax: 1-212-944-1318
E-mail: acmhelp@acm.org

ACM European Service Center
108 Cowley Road
Oxford 0X4 1JF
United Kingdom
Phone: +44-1865-382388
Fax: +44-1865-381388
E-mail: acm_europe@acm.org
URL: http://www.acm.org

Selected ACM titles

Component Software: Beyond Object-Oriented Programming *Clemens Szyperski*

The Object Advantage: Business Process Reeingineering with Object Technology (2nd edn) *Ivar Jacobson, Maria Ericsson, Agneta Jacobson, Gunnar Magnusson*

Object-Oriented Software Engineering: A Use Case Driven Approach *Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard*

Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design *Larry L Constantine, Lucy A D Lockwood*

Bringing Design to Software: Expanding Software Developments to Include Design Terry *Winograd, John Bennett, Laura de Young, Bradley Hartfield*

CORBA Distributed Objects: Using Orbix *Sean Baker*

Software Requirements and Specifications: A Lexicon of Software Practice, Principles and Prejudices *Michael Jackson*

Business Process Implementation: Building Workflow Systems *Michael Jackson, Graham Twaddle*

Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming *Nissim* Francez, *Ira Forman*

Design Patterns for Object-Oriented Software Development *Wolfgang Free*

Software Testing in the Real World: Improving the Process *Ed Kit*

Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing *Tim Koomen, Martin Pol*

Requirements Engineering and Rapid Development: An Object-Oriented Approach *Ian Graham*

Software Test Automation

Effective use of test execution tools

MARK FEWSTER

DOROTHY GRAHAM

ACM Press
New York

Addison-Wesley

An imprint of **Pearson Education**
Harlow, England • London • Reading, Massachusetts
Menlo Park, California • New York
Don Mills, Ontario * Amsterdam • Bonn
Sydney • Singapore • Tokyo * Madrid
San Juan • Milan • Mexico City • Seoul • Taipei

PEARSON EDUCATION LIMITED

Head Office:
Edinburgh Gate
Harlow CM20 2JE
Tel:+44 (0)1279 623623
Fax: +44 (0) 1279 431059

London Office:
128 Long Acre
London WC2E 9AN
Tel:+44 (0)20 7447 2000
Fax: +44 (0)20 7240 5771

Website: *www.awl.com/cseng/*

h'irsl published in Grout liriUiiu 1994

ISBN 0-201-33140-3

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

Library of Congress Cataloging in Publication Data
Fewster, Mark, 1959-
Soltvvarc lest automation / Mark Fcwster, Dorothy Graham.
p. cm.
"ACM Press books" -CIP facing t.p. Includes
bibliographical references and index. ISBN
0-201-33140-3 (pbk. : alk. paper)
1. Computer software-Testing-Automation. I. Graham, Dorothy,
1944- . II. Title. QA76.76.T48F49 1999
005. l'4-dc21 99-35578
CIP

10 9 8 7 6 5 4 3

Typeset by Panlek Arts, Maidstonc, Kent.
Printed and bound in Great Britain by Biddies Ltd, Guildford and King's Lynn.

The Publishers' policy is to use paper manufactured from sustainable forests.

Contents

Foreword

In a better world than ours, there would be magic. If you had a problem, you'd approach a wise old magician who embodied the experience of years of study and practice. You'd give the magician some money. In return, you would receive a magical object to make your problem go away. We don't have magic, but we do have technology. And, as Arthur C. Clarke famously wrote, 'any sufficiently advanced technology is indistinguishable from magic.'

That brings us to test automation. As a tester or test manager, one of your problems is that developers keep changing the product. They introduce new bugs in what used to work. You're responsible for finding those bugs. To do that, you re-execute tests you've already run. The yield (bugs found per test) is low. You need a way to make the cost (human effort) correspondingly low. Only then can you escape the trap of spending so much time re-running old tests that you have no time to create new ones.

So why not give your money to a magician (test automation tool vendor) and receive a magical object (testing tool) that will make the problem go away? Simply create the test once and let the tool take it from there. Why not? The technology isn't sufficiently advanced. In untutored hands, the magic object won't work. Without the magician's touch, it will be dead ... rotting … a destructive distraction from the job of finding bugs.

That's where this book comes in. It will teach you how to.make automated testing tools useful, instead of a big rat hole down which to pour your precious time and budget. It embodies the experience of a fantastic number of people who have suffered through trial and error. Reports about what they've learned have been too scattered: a conference paper here, a few pages in a book there, a conversation over beers somewhere else. Mark Fewster and Dorothy Graham have taken that scattered experience and made it accessible, concise, specific, and systematic.

I wish I'd read this book in 1981. I'd have spent less time trying and much less time erring.

Brian Marick
Principal SQA Practitioner,
Reliable Software Technologies

Preface

What this book is about

This book describes how to structure and build an automated testing regime that will give lasting benefits in the use of test execution tools to automate testing on a medium to large scale. We include practical advice for selecting the right tool and for implementing automated testing practices within an organization.

An extensive collection of case studies and guest chapters are also included, reflecting both good and bad experiences in test automation together with useful advice.

Who is this book for?

This book is intended to be of most help to those starting out on the road to automation, so they do not have to spend time discovering the hard way the most likely problems and how to avoid them. The target audience for this book includes:

- potential and recent purchasers of test execution automation tools;
- those who already have a test execution automation tool but are having problems or are not achieving the benefit they should;
- « people responsible for test automation within an organization;
- anyone who is building an in-house test execution automation tool;
- technical managers who want to insure that test automation provides benefits;
- testers and test managers who want to insure that tests are automated well;
- management consultants and test consultants;
- test tool vendors.

Why should I read this book?

If you want to automate testing (not just automate a few standalone tests) then this book will be useful to you. Specifically:

» if you automate on a small scale, you will pick up some useful tips;

» if you automate hundreds of tests, this book will help you do so efficiently;

« if you automate thousands of tests, you will be more certain of long-term success if you follow the advice given in this book (or have discovered it for yourself).

You do *not* need to read this book:

» if you plan to run all your tests manually, forever;

« if you only hope to automate a few tests, say up to a dozen or so;

» if your tests will only be run once and will never be needed again;

» if you don't care that your test automation actually takes longer and costs more than manual testing.

The more tests you automate, the more benefit you can gain from this book.

Spaghetti tests?

An analogy for the main subject of this book is structured programming. Before structured techniques were used, there was little discipline in software engineering practices. It appeared that all you needed to know was a programming language and how to work the compiler. This led to serious problems with 'spaghetti code,' i.e. code that was difficult to understand, hard to test, and expensive to maintain. The search for a solution led eventually to a disciplined approach, where requirements were analyzed and software designed before the code was written. The benefits of a structured approach included greatly increased maintainability, understandability, and testability of the software produced.

Today, test execution tools are very popular, particularly capture replay tools, but there is little discipline in the design of the test automation in many organizations. The obvious benefits of test execution tools are in automating a tedious but necessary task, that of regression testing. However, there is an assumption that all you need to know is a scripting language and how to work the test execution tool. This approach leads to 'spaghetti tests' that are hard to understand and difficult and expensive to maintain, thereby negating the benefits of automating testing. This book is about 'testware design techniques' that are parallel to the 'software design techniques' of 20 or more years ago.

What this book does not include

We do not include any information about specific tools currently on the market (except where they are mentioned in the case studies in Part 2). This is intentional, for two reasons. First, this book is about techniques that are applicable using any test execution tool, available now or in the near future. This book therefore contains generic principles that are not restricted to any existing specific tools. Second, the test execution tool market is very volatile, so any specific information would soon become out of date.

This book does not include techniques for testing, i.e. for designing good test cases. Although an important topic, it is outside the scope of this book (that's our next project).

This book covers the automation of test execution, not the automatic generation of test inputs, for reasons explained in Chapter 1.

How to read this book

This book is designed so that you can dip into it without having to read all of the preceding chapters. There arc two parts to the book. Part 1 is technical and will give you the techniques you need to construct an efficient test automation regime. Part 2 contains case studies of test automation in a variety of organizations and with varying degrees of success, plus some guest chapters giving useful advice. Any chapter in Part 2 can be read on its own, although there are two sets of paired chapters. A guided tour of the content of the case studies is given in the introduction to Part 2.

We have used some terms in this book that are often understood differently in different organizations. We have defined our meaning of these terms in the Glossary at the end of the book. Glossary terms appear in bold type the first time they are used in the book.

The table on the next page shows our recommendations for the chapters that you may like to read first, depending on your objectives.

Guided tour to Part 1: techniques for automating test execution

Each chapter in Part 1 concludes with a summary, which contains an overview of what is covered in that chapter. Here we outline the main points of each chapter.

Chapter 1 is a general introduction to the content of the book. We discuss testing and the difference between the testing discipline and the subject of this book, and we explain why we have concentrated on test execution automation.

If you have not yet experienced the difference between post-purchase euphoria and the grim reality of what a tool will and will not do for you,

If you are:	*Read these chapters first*
Shopping for a test execution tool	1, 2, and 10
A manager wondering why test automation has failed	1, 2, 7, and 8
Using a test execution tool to automate tests, i.e. you will be writing scripts, etc.	2, 3, 4, and 5 (and then 6, 7, 8, and 9)
Concerned with achieving the best benefits from testing and test automation	1 and 8
Having difficulty persuading other people in your organization to use the tools you already have in place	1 and 11
Wanting to know what other people have done	Any chapter in Part 2
Wanting to know why we think we know the answers to the problems of test automation	12
Having problems with maintenance of automated tests	7, then 3 and 5
Wanting to know what other people advise	22, 24, 26, 27, and 28
Interested in 'failure' stories	19, then 13

Chapter 2 will open your eyes. A simple application ('Scribble') is tested manually and then taken through typical initial stages of the use of a capture replay tool.

Chapters 3-9 contain the technical content of the book. Chapter 3 describes five different scripting techniques. Chapter 4 discusses automated comparison techniques. Chapter 5 describes a practical testware architecture. Chapter 6 covers automation of set-up and clear-up activities. Chapter 7 concentrates on testware maintenance. Chapter 8 discusses metrics for measuring the quality of both testing and of an automation regime. Chapter 9 brings together a number of other important topics.

The next two chapters deal with tool evaluation and selection (Chapter 10) and tool implementation within the organization (Chapter 11).

Acknowledgments

We would like to thank the following people for reviewing early drafts of this book: Yohann Agalawatte, Steve Allott, Stale Amland, Hans Buwalda, Greg Daich, Peter Danson, Norman Fenton, Dave Gelperin, Paul Gerrard, David Hedley, Bill Hctzel, Herb Isenberg, Ed Kit, David Law, Shari Lawrence-Pfleeger, Aidcn Magill, Pat McGee, Geoff Quentin, David Ramsay, Krystyna Rogers, Hans Schaefer, Melanie Smith, Graham Titterington, Otto Vintcr, Jelena Vujatov, Stuart Walker, Robin Webb, and Kerry Zallar.

We are particularly grateful to the authors of the case studies, for taking the time out of their busy schedules to prepare their material. Thanks to Keiron Marsden for bringing up to date the case history in Chapter 12, Marnie Hutcheson for sharing insights and lessons learned, and Clive Bates, Peter Oakley, Paul Herzlich, Simon Mills, Steve Allott, Bob Bartlett, Susan Windsor and Lloyd Roden for chapters on automation in various UK industries. Special thanks to Marnie Hutcheson and Stale Amland for sharing experiences that were not as successful as they might have been, and to all the authors for their honesty about problems encountered. Thanks to two sets of authors who have given us a description and experience story of using what we consider to be good ways of achieving efficient and maintainable automation: Hans Buwalda and Iris Pinkster for the 'Action Words' approach, and Graham Freeburn, Graham Dwyer, and Jim Thomson for the 'RadSTAR' approach. We are particularly grateful to the US authors of the final three chapters: Linda Hayes has kindly given permission to reproduce a number of sections from her *Test Automation Handbook,* Chip Groder has shared his insights in building effective test automation for GUI systems over many years, and Angela Smale gives useful advice based on her experiences of automating the testing for different applications at Microsoft. Thanks to Roger Graham for his support and for writing our example application 'Scribble'. Thanks to Sally Mortimore, our editor, for her support, enthusiasm, and patience over the past three years.

We would also like to thank those who have attended our tutorials and courses on test automation and have helped us to clarify our ideas and the way in which we communicate them.

Some parts of this book first appeared in different forms in the following publications:

The CAST Report, Cambridge Market Intelligence, 1995 (Chapters 10 and 11) *The Journal of Software Testing, Verification and Reliability,* 1991, Sigma Press, now published by Wiley (Chapter 12)
Proceedings of the International Conference on Testing Computer Software, 1991 (Chapter 12)
Proceedings of the EuroSTAR conference, 1993 (Chapter 7).

Please accept our apologies if we have not included anyone in this acknowledgment that we should have.

Mark Fewster and Dorothy Graham, May 1999
http://www.grove.co. uk

Mark Fewster

Mark has 20 years of industrial experience in software engineering, half of this specializing in software testing. He has been a software developer and manager for a multi-platform graphical application vendor, where he was responsible for the implementation of a testing improvement programme and the successful development and implementation of a testing tool which led to dramatic and lasting savings for the company.

Mark spent two years as a consultant for a commercial software testing tool vendor, providing training and consultancy in both test automation and testing techniques.

Since joining Grove Consultants in 1993, Mark has provided consultancy and training in software testing to a wide range of companies. As a consultant, Mark has helped many organizations to improve their testing practices through improved process and better use of testing tools.

Mark serves on the committee of the British Computer Society's Specialist Interest Group in Software Testing (BCS SIGIST). He has also been a member of the Information Systems Examination Board (ISEB) working on a qualification scheme for testing professionals and served on the BCS SIGIST working party that drafted the Software Component Testing Standard BS7925-1. He has published papers in respected journals and is a popular speaker at national and international conferences and seminars.

Dorothy Graham

Dorothy has worked in the area of Software Testing for more than 25 years, first for Bell Telephone Laboratories in New Jersey and then with Ferranti Computer Systems in the UK. Before founding Grove Consultants she worked for the National Computing Centre developing and presenting software engineering training courses.

She originated the *CAST Report* on Computer-Aided Software Testing Tools, published by Cambridge Market Intelligence, and is co-author with Tom Gilb of *Software Inspection* (Addison-Wesley, 1993). She has written articles for a number of technical journals, and is a frequent and popular keynote and tutorial speaker at national and international conferences and seminars.

Dot was Programme Chair for the first EuroSTAR Conference in 1993. She is on the editorial boards of the Journal for *Software Testing, Verification and Reliability* and the *Software Testing & Quality Engineering* magazine, and is a board member for the International Conference on Testing Computer Software (an annual conference in Washington DC USA) and the Information Systems Examination Board for Software Testing (ISEB).

Grave Consultants' Web site: www.grovc.co.uk

Techniques for automating test execution

Trademark notice

Permissions acknowledgment

The publisher and the authors would like to thank the following for permission to reproduce material in this book.

Paul Godsafe for the Experience Report (p. 41) about Squat the robot dog, from his presentation 'Automated testing', at the *British Association of Research Quality Assurance 13th International Conference,* May 1998, Glasgow; Computer Weekly for the Experience Report (p. 61) *Bug in jet engine downs Microsoft's high-flyer,* published on 10 September 1998, Computer Weekly, Quadrant House, The Quandrant, Sutton, Surrey SM2 5AS UK; Herb Isenbcrg, for the Experience Report (p. 97) adapted from 'Flexible testing systems', published in *Dr Dobb's journal,* June 1995, pp. 88-95. Dr Dobb's Journal, P.O. Box 56188, Boulder, CO 80322-6188 USA; Robert Glass for Experience Report (p. 282) 'Testing automation, a horror story', originally published in *The Software Practitioner,* Mar-Apr 1995, published by Computing Trends, 1416 Sare Road, Bloomington, Indiana 47401 USA; Chris Kcmerer (p. 284), for information from 'How the learning curve affects CASE tool adoption', published in *IEEE Software,* May 1992, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855-1331 USA.

The section supplied by Susan Windsor and Bob Bartlett is gratefully received from SIM Group Ltd as a small representation of the many fascinating success stories SIM Group has achieved with automated testing over the years. The fact that SIM Group have consistently made significant differences to projects through the use of automated testing is reason enough for this book to exist.

The Publishers and the Authors wish to thank Angela Smale (nee McAuley) for permission to include details of her experience as a Test Manager at Microsoft Corporation in Chapter 28.

At first glance, it seems easy to automate testing: just buy one of the popular **test execution tools,** record the manual tests, and play them back whenever you want to. Unfortunately, as those who tried it have discovered, it doesn't work like that in practice. Just as there is more to software design than knowing a programming language, there is more to automating testing than knowing a testing tool.

1.1 Introduction

Software must be tested to have confidence that it will work as it should in its intended environment. Software testing needs to be effective at finding any **defects** which are there, but it should also be efficient, performing the tests as quickly and cheaply as possible.

Automating software testing can significantly reduce the effort required for adequate testing, or significantly increase the testing which can be done in limited time. Tests can be run in minutes that would take hours to run manually. The case studies included in this book show how different organizations have been able to automate testing, some saving significant amounts of money. Savings as high as 80% of manual testing effort have been achieved. Some organizations have not saved money or effort directly but their test automation has enabled them to produce better quality software more quickly than would have been possible by manual testing alone.

A mature test automation regime will allow testing at the 'touch of a button' with tests run overnight when machines would otherwise be idle. Automated tests are repeatable, using exactly the same inputs in the same sequence time and again, something that cannot be guaranteed with manual testing. Automated testing enables even the smallest of maintenance changes to be fully tested with minimal effort. Test automation also

eliminates many menial chores. The more boring testing seems, the greater the need for tool support.

This book will explain the issues involved in successfully automating software testing. The emphasis is on technical design issues for automated testware. Testware is the set of files needed for ('automated') testing, including scripts, inputs, expected outcomes, set-up and clear-up procedures, files, databases, environments, and any additional software or utilities used in automated testing. (See the Glossary for definitions of terms used in this book, which appear in bold the first time they are used.)

In this introductory chapter we look at testing in general and the automation of (parts of) testing. We explain why we think test execution and result comparison is more appropriate to automate than test design, and describe the benefits, problems, and limitations of test automation.

A regime is a system of government. This book is about how to set up a regime for test automation. A test automation regime determines, among other things, how test automation is managed, the approaches used in implementing automated tests, and how testware is organized.

1.2 Testing and test **automation are different**

1.2.1 Testing

Testing is a skill. While this may come as a surprise to some people it is a simple fact. For any system there is an astronomical number of possible test cases and yet practically we have time to run only a very small number of them. Yet this small number of test cases is expected to find most of the defects in the software, so the job of selecting which test cases to build and run is an important one. Both experiment and experience have told us that selecting test cases at random is not an effective approach to testing. A more thoughtful approach is required if good test cases are to be developed.

What exactly is a good test case? There are four attributes that describe the quality of a test case; that is, how good it is. Perhaps the most important of these is its defect detection effectiveness, whether or not it finds defects, or at least whether or not it is likely to find defects. A good test case should also be exemplary. An exemplary test case should test more than one thing, thereby reducing the total number of test cases required. The other two attributes are both cost considerations: how economical a test case is to perform, analyze, and debug; and how evolvable it is, i.e. how much maintenance effort is required on the test case each time the software changes.

These four attributes must often be balanced one against another. For example, a single test case that tests a lot of things is likely to cost a lot to perform, analyze, and debug. It may also require a lot of maintenance each time the software changes. Thus a high measure on the exemplary scale is likely to result in low measures on the economic and evolvable scales.

So the skill of testing is not only in ensuring that test cases will find a high proportion of defects, but also ensuring that the test cases are well designed to avoid excessive costs.

1.2.2 Test automation

Automating tests is also a skill but a very different skill from testing. Many organizations are surprised to find that it is more expensive to automate a test than to perform it once manually. In order to gain benefits from test automation, the tests to be automated need to be carefully selected and implemented. Automated quality is independent of test quality.

Whether a **test** is automated or performed manually affects neither its effectiveness nor how exemplary it is. It doesn't matter how clever you are at automating a test or how well you do it, if the test itself achieves nothing then the end result is a test that achieves nothing faster. Automating a test affects only how economic and evolvable it is. Once implemented, an automated test is generally much more economic, the cost of running it being a mere fraction of the effort to perform it manually. However, automated tests generally cost more to create and maintain. The better the approach to automating tests the cheaper it will be to implement them in the long term. If no thought is given to maintenance when tests are automated, updating an entire automated test suite can cost as much, if not more, than the cost of performing all of the tests manually.

Figure 1.1 shows the four quality attributes of a test case in a Keviat diagram. A test case performed manually is shown by the solid lines. When that same test is automated for the first time, it will have become less evolvable and less economic (since it has taken more effort to automate it). After the automated test has been run a number of times it will become much more economic than the same test performed manually.

For an effective and efficient suite of automated tests you have to start with the raw ingredient of a good test suite, a set of tests skillfully designed by a **tester** to exercise the most important things. You then have to apply automation skills to automate the tests in such a way that they can be created and maintained at a reasonable cost.

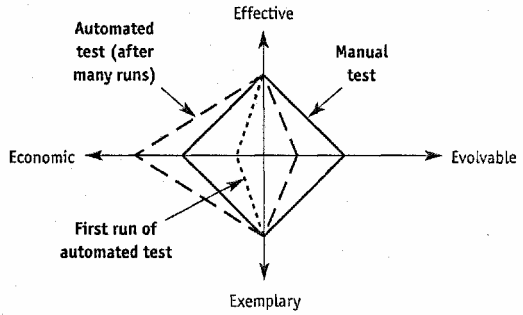


Figure 1.1 The 'goodness' of a test case can be illustrated by considering the four attributes in this Keviat diagram. The greater the measure of each attribute the greater the area enclosed by the joining lines and the better the test case.

The person who builds and maintains the artifacts associated with the use of a test execution tool is the **test automator.** A test automator may or may not also be a tester; he or she may or may not be a member of a test team. For example, there may be a test team consisting of user testers with business knowledge and no technical software development skills. A developer may have the responsibility of supporting the test team in the construction and maintenance of the automated implementation of the tests designed by the test team. This developer is the test automator.

It is possible to have either good or poor quality testing. It is the skill of the tester which determines the quality of the testing.

It is also possible to have either good or poor quality automation. It is the skill of the test automator which determines how easy it will be to add new automated tests, how maintainable the automated tests will be, and ultimately what benefits test automation will provide.

1.3 The V-model

Having made the important distinction between testing and automation, we return to the topic of testing in more detail. Testing is often considered something which is done after software has been written; after all, the argument runs, you can't test something that doesn't exist, can you? This idea makes the assumption that testing is merely test execution, the running of tests. Of course, tests cannot be executed without having software that actually works. But testing activities include more than just running tests.

The V-model of software development illustrates when testing activities should take place. The V-modcl shows that each development activity has a corresponding test activity. The tests at each level exercise the corresponding development activity. The same principles apply no matter what software life cycle model is used. For example, Rapid Application Development (RAD) is a series of small Vs.

The simplified V-model in Figure 1.2 shows four levels of development and testing activity. Different organizations may have different names for each stage; it is important that each stage on the left has a partner on the right, whatever each is called.

The most important factor for successful application of the V-model is the issue of *when* the test cases are designed. The test design activity always finds defects in whatever the tests are designed against. For example, designing acceptance test cases will find defects in the requirements, designing system test cases will find defects in the functional specification, designing integration test cases will find defects in the design, and designing unit test cases will find defects in the code. If test design is left until the last possible moment, these defects will only be found immediately before those tests would be run, when it is more expensive to fix them.

Test design does not have to wait until just before tests are run; it can be done at any time after the information which those tests are based on becomes available. Then the effect of finding defects is actually beneficial rather than destructive, because the defects can be corrected before they are propagated.

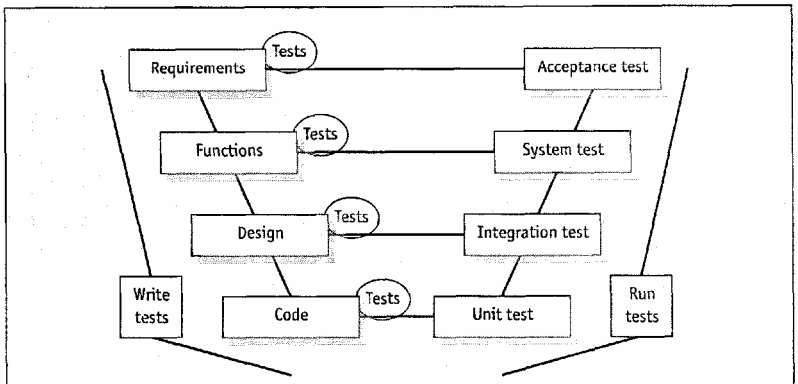


Figure 1.2 The V-model showing early test design.

Of course, the tests cannot be run until the software has been written, but they can be written early. The tests are actually run in the reverse order to their writing, e.g. unit tests are written last but are run first.

1.4 Tool support for life-cycle testing

Tool support is available for testing in every stage of the software development life cycle; the different types of tools and their position within the life cycle are shown in Figure 1.3, using alternative names for development stages.

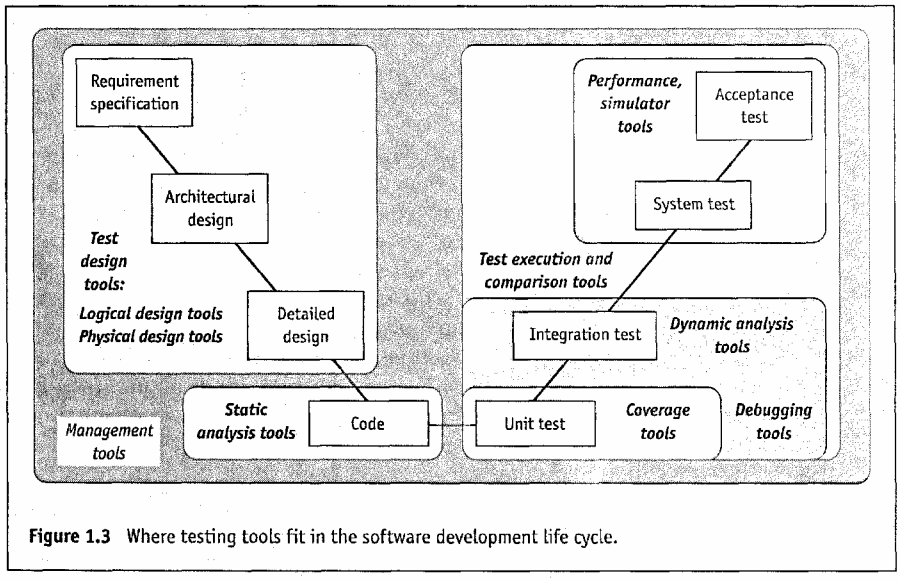


Figure 1.3 Where testing tools fit in the software development life cycle.

Test design tools help to derive **test inputs or test data.** Logical design tools work from the logic of a specification, an interface or from code, and are sometimes referred to as test case generators. Physical design tools manipulate existing data or generate test data. For example, a tool that can extract random records from a database would be a physical design tool. A tool that can derive test inputs from a specification would be a logical design tool. We discuss logical design tools more fully in Section 1.8.

Test management tools include tools to assist in test planning, keeping track of what tests have been run, and so on. This category also includes tools to aid traceability of tests to requirements, designs, and code, as well as defect tracking tools.

Static analysis tools analyze code without executing it. This type of tool detects certain types of defect much more effectively and cheaply than can be achieved by any other means. Such tools also calculate various metrics for the code such as McCabe's cyclomatic complexity, Halstead metrics, and many more.

Coverage tools assess how much of the software under test has been exercised by a set of tests. Coverage tools are most commonly used at unit test level. For example, branch coverage is often a requirement lor testing safety-critical or safety-related systems. Coverage tools can also measure the coverage of design level constructs such as call trees.

Debugging tools are not really testing tools, since debugging is not part of testing. (Testing identifies defects, debugging removes them and is therefore a development activity, not a testing activity.) However, debugging tools are often used in testing, especially when trying to isolate a low-level defect. Debugging tools enable the developer to step through the code by executing one instruction at a time and looking at the contents of data locations.

Dynamic analysis tools assess the system while the software is running. For example, tools that can detect memory leaks are dynamic analysis tools. A memory leak occurs if a program does not release blocks of memory when it should, so the block has 'leaked' out of the pool of memory blocks available to all programs. Eventually the faulty program will end up 'owning' all of the memory; nothing can run, the system 'hangs up' and must be rebooted (in a non-protected mode operating system).

Simulators are tools that enable parts of a system to be tested in ways which would not be possible in the real world. For example, the meltdown procedures for a nuclear power plant can be tested in a simulator.

Another class of tools have to do with what we could call capacity testing. **Performance testing** tools measure the time taken for various events. For example, they can measure response times under typical or load conditions. **Load testing** tools generate system traffic. For example, they may generate a number of transactions which represent typical or maximum levels. This type of tool may be used for **volume and stress testing.**

Test execution and comparison tools enable tests to be executed automatically and the **test outcomes** to be compared to **expected outcomes.** These tools are applicable to test execution at any level: unit, integration,

system, or acceptance testing. Capture replay tools are test execution and comparison tools. This is the most popular type of testing tool in use, and is the focus of this book.

1.5 The promise of test automation

Test automation can enable some testing tasks to be performed far more efficiently than could ever be done by testing manually. There are also other benefits, including those listed below.

1. Run existing (regression) tests on a new version of a program. This is perhaps the most obvious task, particularly in an environment where many programs are frequently modified. The effort involved in perform ing a set of regression tests should be minimal. Given that the tests already exist and have been automated to run on an earlier version of the program, it should be possible to select the tests and initiate their execution with just a few minutes of manual effort.
2. Run more tests more often. A clear benefit of automation is the ability to run more tests in less time and therefore to make it possible to run them more often. This will lead to greater confidence in the system. Most people assume that they will run the same tests faster with automation. In fact they tend to run more tests, and those tests are run more often.
3. Perform tests which would be difficult or impossible to do manually. Attempting to perform a full-scale live test of an online system with say 200 users may be impossible, but the input from 200 users can be simu lated using automated tests. By having end users define tests that can be replayed automatically, user scenario tests can be run at any time even by technical staff who do not understand the intricacies of the full busi ness application.

   When testing manually, expected outcomes typically include the obvi- ous things that are visible to the tester. However, there are attributes that should be tested which are not easy to verify manually. For exam- ple a graphical user interface (GUI) object may trigger some event that does not produce any immediate output. A test execution tool may be able to check that the event has been, triggered, which would not be possible to check without using a tool.
4. Better use of resources. Automating menial and boring tasks, such as repeatedly entering the same test inputs, gives greater accuracy as well as improved staff morale, and frees skilled testers to put more effort into designing better test cases to be run. There will always be some testing which is best done manually; the testers can do a better job of manual testing if there are far fewer tests to be run manually.

   Machines that would otherwise lie idle overnight or at the weekend can be used to run automated tests.
5. Consistency and repeatability of tests. Tests that are repeated automati cally will be repeated exactly every time (at least the inputs will be; the

outputs may differ due to timing, for example). This gives a level of consistency to the tests which is very difficult to achieve manually.

The same tests can be executed on different hardware configurations, using different operating systems, or using different databases. This gives a consistency of cross-platform quality for multi-platform products which is virtually impossible to achieve with manual testing.

The imposition of a good automated testing regime can also insure consistent standards both in testing and in development. For example, the tool can check that the same type of feature has been implemented in the same way in every application or program.

6. Reuse of tests. The effort put into deciding what to test, designing the tests, and building the tests can be distributed over many executions of those tests. Tests which will be reused are worth spending time on to make sure they are reliable. This is also true of manual tests, but an automated test would be reused many more times than the same test repeated manually.
7. Earlier time to market. Once a set of tests has been automated, it can be repeated far more quickly than it would be manually, so the testing elapsed time can be shortened (subject to other factors such as avail ability of developers to fix defects).
8. Increased confidence. Knowing that an extensive set of automated tests has run successfully, there can be greater confidence that there won't be any unpleasant surprises when the system is released (providing that the tests being run are good tests!).

In summary, more thorough testing can be achieved with less effort, giving increases in both quality and productivity.

1.6 Common problems of test automation

There are a number of problems that may be encountered in trying to automate testing. Problems which come as a complete surprise are usually more difficult to deal with, so having some idea of the type of problems you may encounter should help you in implementing your own automation regime. Most problems can be overcome, and this book is intended to help you deal with them. We describe some of the more common problems below.

1. Unrealistic expectations. Our industry is known for latching onto any new technical solution and thinking it will solve all of our current problems. Testing tools are no exception. There is a tendency to be optimistic about what can be achieved by a new tool. It is human nature to hope that this solution will at last solve all of the problems we are currently experiencing. Vendors naturally emphasize the benefits and successes, and may play down the amount of effort needed to achieve lasting benefits. The effect of optimism and salesmanship together is to encourage unrealistic expectations. If management expectations are unrealistic,

then no matter how well the tool is implemented from a technical point of view, it will not meet expectations.

2. Poor testing practice. If testing practice is poor, with poorly organized tests, little or inconsistent documentation, and tests that are not very good at finding defects, automating testing is not a good idea. It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing.

   Automating chaos just gives faster chaos.

3. Expectation that automated tests will find a lot of new defects. A test is most likely to find a defect the first time it is run. If a test has already run and passed, running the same test again is much less likely to find a new defect, unless the test is exercising code that has been changed or could be affected by a change made in a different part of the software, or is being run in a different environment.

   Test execution tools are 'replay' tools, i.e. regression testing tools. Their use is in repeating tests that have already run. This is a very useful thing to do, but it is not likely to find a large number of new defects, particularly when run in the same hardware and software environment as before.

   Tests that do not find defects are not worthless, even though good test design should be directed at trying to find defects. Knowing that a set of tests has passed again gives confidence that the software is still working as well as it was before, and that changes elsewhere have not had unforeseen effects.

4. False sense of security. Just because a test suite runs without finding any defects, it does not mean that there are no defects in the software. The tests may be incomplete, or may contain defects themselves. If the expected outcomes are incorrect, automated tests will simply preserve those defective results indefinitely.

5. Maintenance of automated tests. When software is changed it is often necessary to update some, or even all, of the tests so they can be re-run successfully. This is particularly true for automated tests. Test maintenance effort has been the death of many test automation initiatives. When it takes more effort to update the tests than it would take to re-run those tests manually, test automation will be abandoned. One of the purposes of this book is to help you make sure that your test automation initiative does not fall victim to high maintenance costs.

6. Technical problems. Commercial test execution tools are software products, sold by vendor companies. As third-party software products, they are not immune from defects or problems of support. It is perhaps a double disappointment to find that a testing tool has not been well tested, but unfortunately, it does happen.

   Interoperability of the tool with other software, either your own applications or third-party products, can be a serious problem. The technological environment changes so rapidly that it is hard for the

vendors to keep up. Many tools have looked ideal on paper, but have simply failed to work in some environments.

The commercial test execution tools are large and complex products, and detailed technical knowledge is required in order to gain the best from the tool. Training supplied by the vendor or distributor is essential for all those who will use the tool directly, particularly the test automa-tor(s) (the people who automate the tests).

In addition to technical problems with the tools themselves, you may experience technical problems with the software you are trying to test. If software is not designed and built with testability in mind, it can be very difficult to test, either manually or automatically. Trying to use tools to test such software is an added complication which can only make test automation even more difficult.

7. Organizational issues. Automating testing is not a trivial exercise, and it needs to be well supported by management and implemented into the culture of the organization. Time must be allocated for choosing tools, for training, for experimenting and learning what works best, and for promoting tool use within the organization.

   An automation effort is unlikely to be successful unless there is one person who is the focal point for the use of the tool, the tool 'champion.' Typically, the champion is a person who is excited about automating testing, and will communicate his or her enthusiasm within the company. This person may be involved in selecting what tool to buy, and will be very active in promoting its use internally. More detail on the role of the champion is given in Chapters 10 and 11.

   Test automation is an infrastructure issue, not just a project issue. In larger organizations, test automation can rarely be justified on the basis of a single project, since the project will bear all of the start-up costs and teething problems and may reap little of the benefits. If the scope of test automation is only for one project, people will then be assigned to new projects, and the impetus will be lost. Test automation often falls into decay at precisely the time it could provide the most value, i.e. when the software is updated. Standards are needed to insure consistent ways of using the tools throughout the organization. Otherwise every group may develop different approaches to test automation, making it difficult to _ transfer or share automated tests and testers between groups.

   Even a seemingly minor administrative issue such as having too few licenses for the people who want to use the tool can seriously impact the success and the cost of a test automation effort.

   Perceptions of work effort may also change. If a test is run overnight, then when the testers arrive in the morning, they will need to spend some time looking through the results of the tests. This test analysis time is now clearly visible as a separate activity. When those tests were run manually, this test analysis time was embedded in the test execution activity, and was therefore not visible.

   Whenever a new tool (or indeed any new process) is implemented, there are inevitably adjustments that need to be made to adapt to new

ways of working, which must be managed. Chapter 11 discusses ways in which you can insure successful implementation of test automation within an organization.

EXPERIENCE REPORT: SPURIOUS DEFECT PRESERVED

A client had a geographical information system that produced maps showing various underground structures, both artificial and natural. The company had an automated testing system that it was very proud of, as it had been in continuous use for over three years. Every release of the software had to pass the automated tests before it could be released.

The client was somewhat mystified at an error reported by a customer, where there were structures shown on the map produced by the software that should not have been there. Further investigation revealed that three spurious circles were included in the expected outcome for that particular map. The automated tests had insured that the defect had been preserved over all releases of the software for three years.

1.7 Test activities

In this section, we describe testing activities, since these are the activities that we may want to automate. There is wide variation in the way in which these activities are carried out in different organizations. Some organizations will perform all of these activities formally. Others may be informal to the point of being almost chaotic. In any case, the core activities are still carried out more or less in the sequence described in Figure 1.4.

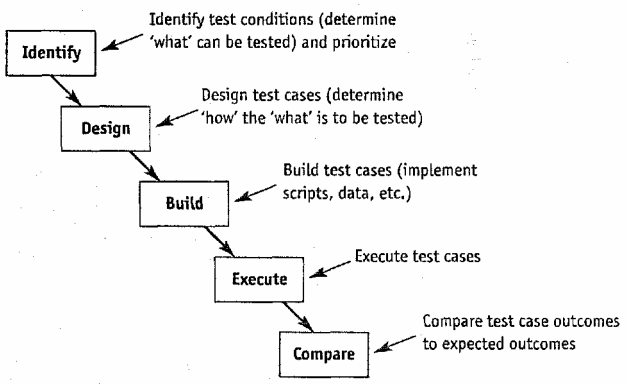


Figure 1.4 The test process comprises a number of distinct activities which may be considered the development life cycle of test cases.

In an ideal world, testing should start with the setting of organizational test objectives and test policy, the setting of test strategies to meet test objectives, and policies and strategies for individual projects. Test planning at the management level would include estimating the time required for all test activities, scheduling and resourcing of test activities, monitoring the progress of testing, and taking any actions required to keep the entire test effort on track. These high-level activities should be done at the beginning of a project and continued throughout its development.

Test cases will typically be developed for various functional areas of the system to be tested. Each of these test cases will go through five distinct development activities (not including re-testing). The five test development activities are sequential for any one test case, i.e. the test conditions must be identified before a test case for those conditions can be designed, a test case must be designed before it can be built, built before it can be run, and run before its results can be compared. The five steps may be done formally, with documentation at each step, or all five steps may be done informally when the tester sits down to test.

Note that there arc many variations of terminology in testing. We have defined what we mean by each term in the Glossary.

1.7.1 Identify test conditions

The first activity is to determine 'what' can be tested and ideally to prioritize these test conditions. A **test condition** is an item or event that could be verified by a test. There will be many different test conditions for a system, and for different categories of test, such as functionality tests, performance tests, security tests, etc.

Testing techniques help testers derive test conditions in a rigorous and systematic way. A number of books on testing describe testing techniques such as **equivalence partitioning, boundary value analysis,** cause-effect graphing, and others. A bibliography about books on testing and other topics can be found on the authors' web site (http://www.grove.co.uk). These include Kit (1995), Marick (1995), Beizer (1990, 1995), Kaner *et al.* (1993), Myers (1979), and Hetzel (1988).

Test conditions are descriptions of the circumstances that could be examined. They may be documented in different ways, including brief sentences, entries in tables, or in diagrams such as flow graphs.

Note that the activity of identifying test conditions is best done in parallel with the associated development activities, i.e. on the left-hand side of the V-model.

1.7.2 Design test cases

Test case design determines how the 'what' will be tested. A **test case** is a set of tests performed in a sequence and related to a **test objective,** i.e. the reason or purpose for the tests. Test case design will produce a number of tests comprising specific input values, expected outcomes, and any other information needed for the test to run, such as environment prerequisites.

Note that the expected outcomes include things that should be output or created, things that should be changed or updated (in a database, for example), things that should not be changed, and things that should be deleted. The set of expected outcomes could be of considerable size.

An example test case is given in Table 1.1. Note that the tester performing this test case would need to understand the purchase order system at least to the extent that he or she would know how to create a new order, verify a purchase order, print a new orders report, and so on.

The following three test conditions are exercised by this test case:

- order created for a single item (VB10);
- order quantity of 100 (VB23);
- e order cancelled (V8).

The tags in parentheses at the end of each condition are used for cross-referencing between test conditions and test cases. The tags used here represent test conditions derived using equivalence partitioning and boundary value analysis. VB10 is the 10th Valid Boundary, VB23 is the 23rd Valid Boundary, and V8 is the 8th Valid equivalence partition.

Table 1.1 An example test case. This comprises five steps (each of which may be considered a single 'test') and exercises three different test conditions.

Test case: POS1036

Prerequisites:
- logged into the purchase order system as a data entry clerk with the main menu displayed;
- database system must contain the standard Data Set;
- there must be no other new purchase order activity on the system.

Step	*Input*	*Expected outcome*	*Jest conditions*
1	Create a new order for any one standard order item, setting order quantity to exactly 100	Order confirmation message displayed	VB10 VB23
2	Confirm the order	Purchase order printed with correct details	VB10 VB23
3	Print a new orders report	New orders report printed showing just this one new order	VB10 VB23
4	Cancel the order	Purchase order cancellation notice printed with correct details	V8
5	Print a new orders report	Report printed showing no outstanding purchase orders	V8

This example test case is taken out of a larger set of test cases. It is assumed that we have already tested other aspects such as the correct printing of purchase orders. In this example test case, we are therefore using some 'trusted' functionality in order to test different aspects. In the last column we have noted the condition tags covered. Steps 2 and 3 also exercise the same tags as step 1, and step 4 and step 5 exercise V8.

Note that this style of test case is what we will define in Chapter 2 as a Vague manual test script.'

Each test should specify the **expected outcome.** If the expected outcome is not specified in advance of it being run, then in order to check whether or not the software is correct, the first **actual outcome** should be carefully examined and verified. This will, of course, require a tester with adequate knowledge of the software under test to make a correct assessment of the outcome. If correct, it can then become the future expected outcome, i.e. the outcome to which future actual outcomes from this test can be automatically compared. This approach is referred to as **reference testing.** The validated expected outcome is sometimes referred to as the **golden version.**

Test case design (ideally with expected outcomes predicted in advance) is best done in parallel with the associated development activities on the left-hand side of the V-model, i.e. before the software to be tested has been built.

1.7.3 Build the test cases

The test cases are implemented by preparing test scripts, test inputs, test data, and expected outcomes. A **test script** is the data and/or instructions with a formal syntax, used by a test execution automation tool, typically held in a file. A test script can implement one or more test cases, navigation, **set-up** or **clear-up** procedures, or verification. A test script may or may not be in a form that can be used to run a test manually (a manual test script is a **test procedure).** The test inputs and expected outcome may be included as part of a script, or may be outside the script in a separate file or database.

The preconditions for the test cases must be implemented so that the tests can be executed. For example, if a test uses some data from a file or database, that file or database must be initialized to contain the information that the test expects to find.

A test case may require special hardware or software to be available, for example a network connection or a printer. This would also form part of the environment needed to be set up before the test case could be run.

The expected outcomes may be organized into files for automation tools to use. For manual testing, they may simply be notes on the manual test procedure or script. Setting up expected outcomes for automated comparison may be considerably more complex than setting up expected outcomes for manual testing. The tools require everything to be spelled out in great detail, so much more rigor is required than for manual testing.

Any aspects of the test building activity that can be prepared in advance, i.e. on the left-hand side of the V-model, will save time later.

1.7.4 Execute the test cases

The software under test is executed using the test cases. For manual testing, this may consist of the testers sitting down and possibly following a printed manual procedure. They would enter inputs, observe outcomes, and make notes about any problems as they go. For automated testing, this may involve starting the test tool and telling it which test cases to execute.

Test execution can only be done after the software exists, i.e. it is an activity on the right-hand side of the V-model.

1.7.5 Compare test outcomes to expected outcomes

The actual outcome of each test must be investigated to see if the software being tested worked correctly. This may be informal confirmation of what the tester expects to see, or it may be a rigorous and detailed comparison of the exact actual outcomes with expected outcomes. The comparison of some outcomes such as messages sent to a screen can be performed while the test is being executed. Other outcomes, such as a change to database records, can only be compared after the test case execution has completed. An automated test may need to use a combination of these two approaches. See Chapter 4 for more about comparison techniques.

The assumption is that if the actual and expected outcomes are the same, then the software has passed the test; if they are different, then the software has failed the test. This is actually an oversimplification. All we can say is that if the actual and expected outcomes do not match, then something needs to be investigated. It may be that the software is incorrect, or it could be that the test was run in the wrong sequence, the expected outcomes were incorrect, the test environment was not set up correctly, or the test was incorrectly specified.

There is a difference between comparing and verifying; a tool can compare but cannot verify. A tool can compare one set of test outcomes to another, and can flag any differences between the two. But the tool cannot say whether or not the outcomes are correct - this is verification, and is normally done by testers. It is the testers who confirm or insure that the results being compared arc in fact correct. In some special circumstances, it is possible to automate the generation of expected outcomes but this is not the case for most of the industrial testing done using commercial test execution tools. See Poston (1996) for a discussion of automated oracles, reversers, and referencers.

1.8 Automate test design?

1.8.1 Activities amenable to automation

As shown in Figure 1.5, the first two test activities, identify test conditions and design test cases, are mainly intellectual in nature. The last two activities, execute test cases and compare test outcomes, are more clerical in nature. It is the intellectual activities that govern the quality of the test cases. The clerical activities are particularly labor intensive and are therefore well worth automating.

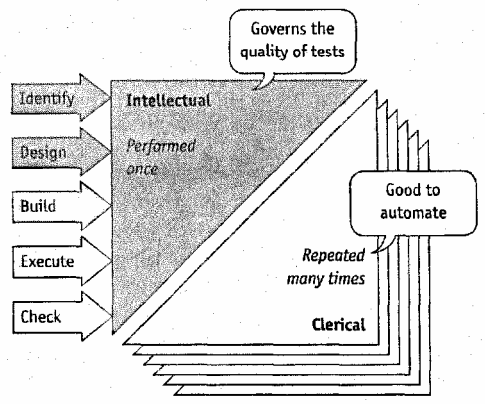


Figure 1.5 The testing process breaks down into five distinct activities. The first two are predominantly intellectual while the last two are predominantly clerical. The intellectual activities govern the quality of the final test cases but as a whole are performed only once. The clerical activities are usually performed many times, making them ideal candidates for automation.

In addition, the activities of test execution and comparison are repeated many times, while the activities of identifying test conditions and designing test cases are performed only once (except for rework due to errors in those activities). For example, if a test finds an **error** in the software, the repeated activities after the fix will be test execution and comparison. If a test fails for an environmental reason such as incorrect test data being used, the repeated test activities would be test building, execution, and comparison. If tests are to be run on different platforms, then those same three activities would be repeated for each platform. When software is changed, regression tests are run to insure that there are no side effects. A regression test will repeat the test execution and comparison activities (and possibly also test building). Activities that are repeated often are particularly good candidates for automation.

All of the test activities can be performed manually, as human testers have been doing for many years. All of the test activities can also benefit from tool support to some extent, but we believe it is in automating the latter test activities where there is most to gain.

The focus of this book is on the automation of the test execution and comparison activities. However, we cannot ignore the automation of test design activities in a book on test automation, since the output of such tools may (or may not) be in a format that can easily be used by a test execution tool. The next section discusses different types of test design automation tools. (Skip to Section 1.9 if this topic does not interest you.)

1.8.2 Automating test case design

Can the activities of test case design be automated? There are a number of ways in which testing tools can automate parts of test case design. These tools are sometimes called test input generation tools and their approach is useful in some contexts, but they will never completely replace the intellectual testing activities.

One problem with all test case design approaches is that the tool may generate a very large number of tests. Some tools include ways of minimizing the tests generated against the tester's specified criteria. However the tool may still generate far too many tests to be run in a reasonable time. The tool cannot distinguish which tests are the most important; this requires creative intelligence only available from human testers. A tool will never be able to answer questions such as: 'If we only have time to test 30% of what we could test, which are the most important test cases to run?' Even if the testing is automated, there may still be insufficient time to run 100% of the automated tests.

All test generation tools rely on algorithms to generate the tests. The tools will be more thorough and more accurate than a human tester using the same algorithm, so this is an advantage. However, a human being will think of additional tests to try, may identify aspects or requirements that are missing, or may be able to identify where the specification is incorrect based on personal knowledge. The best use of test generation tools is when the scope of what can and cannot be done by them is fully understood. (Actually that applies to the use of any tool!)

We will look at three types of test input generation tools, based on code, interfaces, and specifications.

1.8,2.1 Code-based test input generation

Code-based test input generation tools, shown in Figure 1.6, generate test inputs by examining the structure of the software code itself. A path through the code is composed of segments determined by branches at each decision point. A profile of the logical conditions required for each path segment can therefore be produced automatically. This is useful in conjunction with coverage measurement. For example, some coverage tools can identify logical path segments that have not been covered.

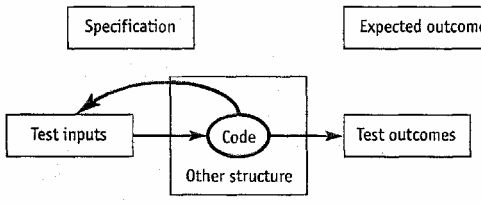


Figure 1.6 Code-based test input generation.

This approach generates test inputs, but a test also requires expected outcomes to compare against. Code-based test case design cannot tell whether the outcomes produced by the software are the correct values - it is the specification for the code that tells you what it should do. So this approach is incomplete, since it cannot generate expected outcomes. A **test oracle** is the source of what the correct outcome should be; code-based test input generation has no test oracle.

Another problem with this approach is that it only tests code that exists and cannot identify code that is missing. It is also testing that 'the code does what the code does.' This is generally not the most useful kind of testing, since we normally want to test that 'the code does what it should do.' (We once saw a defect report dismissed with the phrase: 'software works as coded' (not 'as specified').) This approach cannot find specification defects or missing specifications.

Another type of code-based test design automation can generate tests to satisfy weak mutation testing criteria. **Mutation testing** is where the code or an input is changed in some small way to see if the system can correctly deal with or detect this slightly mutated version. This is used to assess the fault tolerance of systems and the adequacy of test suites. For more information, see Voas and McGraw (1998).

1.8.2.2 Interface-based test generation

An **interface-based test** input generation tool, shown in Figure 1.7, is one that can generate tests based on some well-defined interface, such as a GUI or a web application. If a screen contains various menus, buttons, and check boxes, a tool can generate a test case that visits each one. For example, a tool could test that a check box has a cross in it when checked and is empty when not checked (similarly fundamental tests can be automated on other graphical interactors). Other examples include selecting each field and seeing if *Help* works, trying to edit data in all display-only fields, performing a spell check in help text and legends, and checking that something pops up when each menu item is selected.

A similar function is found in tools to test internet and intranet pages. The tool can activate each link on a World Wide Web page, and then do the same for each of the pages linked to, recursively.

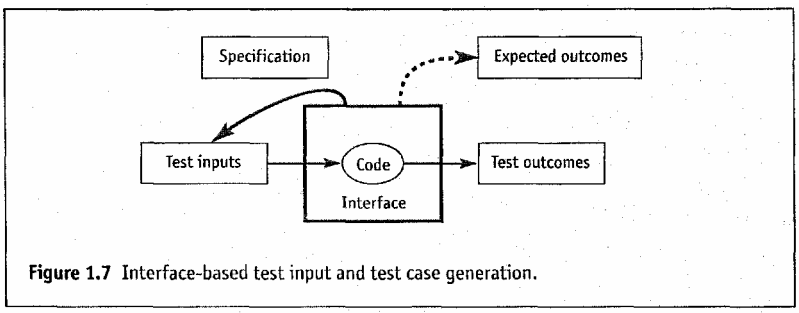


Figure 1.7 Interface-based test input and test case generation.

An interface-based approach is useful in identifying some types of defect (for example, any web page links that do not work can be identified). An expected outcome can be partially generated, but only in a very general sense and only negatively, i.e. a link should be there (correct result), not broken (incorrect result). This type of automated test generation cannot tell you whether working links are in fact linked to the right place.

This approach can therefore perform parts of the test design activity, generating test inputs (one test for each interface element identified by the tool), and can be a partial oracle, which enables some errors to be identified. It can be very useful in performing 'roll-call' testing, i.e. checking that everything which should be there is there. This type of testing is tedious to do manually, but is often very important to do thoroughly.

Tool support of this nature will probably increase in the future, and may help to free testers from tedious tasks so that they have more time to perform the intellectual testing activities.

1.8.2.3 Specification-based test generation

Specification-based test tools can generate test inputs and also expected outputs, provided that the specification is in a form that can be analyzed by the tool. This is shown in Figure 1.8. The specification may contain structured natural language constructs, such as business rules, or it may contain technical data such as states and transitions. Tools can derive tests from object-oriented specifications, if they are sufficiently rigorous. Hence these tools do have a test oracle.

For example, if the allowable ranges for an input field are rigorously defined, a tool can generate boundary values, as well as sample values in valid and invalid equivalence classes. For more information see Poston (1996).

Some specification-based tools can derive tests from structured English specifications, or from cause-effect graphs. This can identify some types of specification defect such as ambiguities and omissions.

A benefit of a specification-based approach is that the tests are looking at what the software should do, rather than what it does do, although only within the limited scope of the existing tool-examinable specification. The more tedious it is to derive test cases from a specification, the higher the potential for this type of tool.

Expected outcomes can be generated if they are stored in the specification, assuming that the stored specification is actually correct.

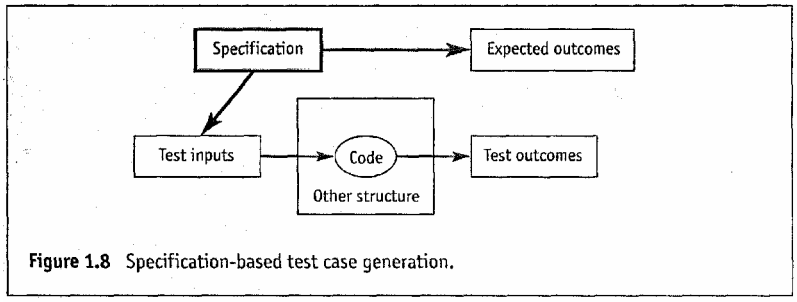


Figure 1.8 Specification-based test case generation.

1.8.2.4 Summary of test design automation

Summary of the benefits of automated test case generation:

» automates tedious aspects of test case design, such as activating every menu item or calculating boundary values from known data ranges;

« can generate a complete set of test cases with respect to their source (code, interface, or specification);

» can identify some types of defect, such as missing links, non-working window items, or software that does not conform to a stored specification.

Summary of the limitations of automated test case generation:

- code-based methods do not generate expected outcomes;
- interface-based methods can only generate partial expected outcomes;

« code-based and interface-based methods cannot find specification defects;

» specification-based methods depend on the quality of the specification;

» all methods may generate too many tests to be practical to run;

» human expertise is still needed to prioritize tests, to assess the usefulness of the generated tests, and to think of the tests that could never be generated by any tool.

It is our contention that the automation of the intellectual part of testing is much more difficult to do, and cannot be done as fully as automation of the more clerical part of testing. When we refer to 'automating software testing' in the rest of this book, we mean the automation of test execution and comparison activities, not the automation of test design or generation of test inputs.

1.9 The limitations of automating software testing

As with all good things, nothing comes without a price. Automated testing is not a panacea, and there are limitations to what it can accomplish.

1.9.1 Does not replace manual testing

It is not possible, nor is it desirable, to automate all testing activities or all tests. There will always be some testing that is much easier to do manually than automatically, or that is so difficult to automate it is not economic to do so. Tests that should probably not be automated include:

» tests that are run only rarely. For example if a test is run only once a year, then it is probably not worth automating that test;

» where the software is very volatile. For example, if the user interface and functionality change beyond recognition from one version to the next, the effort to change the automated tests to correspond is not likely to be cost effective;

- tests where the result is easily verified by a human, but is difficult if not impossible to automate; for example, the suitability of a color scheme, the esthetic appeal of a screen layout, or whether the correct audio plays when a given screen object is selected;
- tests that involve physical interaction, such as swiping a card through a card reader, disconnecting some equipment, turning power off or on.

Not all manual tests should be automated - only the best ones or the ones that will be re-run most often. A good testing strategy will also include some **exploratory** or **lateral testing,**[1] which is best done manually, at least at first. When software is unstable, manual testing will find defects very quickly.

1.9.2 Manual tests find more defects than automated tests

A test is most likely to reveal a defect the first time it is run. If a test case is to be automated, it first needs to be tested to make sure it is correct. (There is little point in automating defective tests!) The test of the test case is normally done by running the test case manually. If the software being tested has defects that the test case can reveal, it will be revealed at this point, when it is being run manually.

James Bach reported that in his (extensive) experience, automated tests found only 15% of the defects while manual testing found 85%.

Source: Bach, 1997

Once an automated test suite has been constructed, it is used to re-run tests. By definition, these tests have been run before, and therefore they are much less likely to reveal defects in the software this time. The test execution tools are not 'testing' tools, they are 're-testing' tools, i.e. regression testing tools.

1.9.3 Greater reliance on the quality of the tests

A tool can only identify differences between the actual and expected outcomes (i.e. compare). There is therefore a greater burden on the task of verifying the correctness of the expected outcomes in automated testing. The tool will happily tell you that all your tests have passed, when in fact they have only matched your expected outcomes.

It is therefore more important to be confident of the quality of the tests that are to be automated. Testware can be reviewed or inspected to insure

[1] Testing should be systematic, rigorous, and thorough in order to be effective at finding defects. However, there is also a valid role in testing for a more intuitive approach to supplement a rigorous approach. This is referred to in most testing books as '**error guessing.**' We prefer to call it 'lateral testing,' after the latcrial thinking techniques of Edward DC Bono.

its quality. Inspection is the most powerful review technique, and is very effective when used on test documentation. This technique is described in Gilb and Graham (1993).

1.9.4 Test automation does not improve effectiveness

Automating a set of tests does not make them any more effective (or exemplary) than those same tests run manually. Automation can eventually improve the efficiency of the tests; that is, how much they cost to run and how long they take to run. But it is likely that automation will adversely affect the evolvability of the test, as discussed in Section 1.2 and shown in Figure 1.1.

1.9.5 Test automation may Limit software development

Automated tests arc more 'fragile' than manual tests. They may be broken by seemingly innocuous changes to software. The techniques described in this book will help you produce automated tests that are less fragile than they would be otherwise, but they will always be more vulnerable to changes in the application software than manual tests.

Because automated tests take mor.e effort to set up than manual tests, and because they require effort to maintain, this in itself may restrict the options for changing or enhancing software systems or applications. Software changes with a high impact on the automated testware may need to be discouraged for economic reasons.

1.9.6 Tools have no imagination

A tool is only software, and therefore can only obediently follow instructions. Both a tool and a tester can follow instructions to execute a set of test cases, but a human tester, given the same task, will perform it differently. For example, if a human tester is given the task of running a prepared test procedure, he or she may follow that procedure to begin with, and will check that the actual outcomes are correct. However, the tester may realize that although the software conforms to the expected outcomes, both are wrong. The human tester can also use his or her creativity and imagination to improve the tests as they are performed, either by deviating from the plan or preferably by noting additional things to test afterwards.

Another way in which human testers are superior to testing tools is when unexpected events happen that are not a part of the planned sequence of a test case. For example, if a network connection is lost and has to be re-established, a human tester will cope with the problem in the middle of the test, and do whatever is necessary. Sometimes, the tester will do this without even realizing that the test has deviated from the plan. However an unexpected event can stop an automated test in its tracks. Of course, the tools can be programmed to cope with some types of event, but

just as astronauts have discovered, there is nothing like human ingenuity to overcome problems.

Summary

Test tools aren't magic - but, properly implemented, they can work wonders!

Source: Hayes, 1995

Test automation is not the same as testing, and the skills needed to be a good test automator are not the same skills needed to be a good tester.

Testing should fit into the software development process at every stage. Test cases are best identified and designed early (on the left-hand side of the V-model), but can only be executed and compared after the software is available for testing (the right-hand side of the V-model).

Tool support is available for all types of test activities throughout the development life cycle, though none of them can (or will ever be able to) make any testing activity completely automatic.

The automation of testing holds great promise and can give significant benefits, such as repeatable consistent regression tests that are run more often, testing of attributes that are difficult to test manually, better use of resources, reuse of tests, earlier time to market, and increased confidence. However, many problems typically arise when attempting to automate testing, including unrealistic expectations, poor testing practices, a false sense of security, maintenance costs, and other technical and organizational problems.

Although there are ways to support test case design activities, we consider that automating the more clerical and tedious aspects of testing generally gives more benefit than trying to automate the more intellectual test activities. This book therefore concentrates on the test activities of executing test cases and comparing outcomes. This is what we mean by automating software testing.

Test automation has its limitations. It does not replace manual testing, and manual testing will find more defects than automated testing. There is a greater reliance on the correctness of expected outcomes. Test automation does not improve test effectiveness, and can limit software development options. The testing tools have no imagination and are not very flexible. However, test automation can significantly increase the quality and productivity of software testing.

Capture replay is not test automation

If you don't know better, it is tempting to think that the quickest way to automate testing is to simply turn on the record facility of a capture replay tool when the tester sits down to test. The recording can then be replayed by the tool, thereby repeating the test exactly as it was performed manually. Unfortunately, many people also believe that this is all there is to test automation. We will show why this is far from the truth.

This chapter describes the way test automation typically evolves when a capture replay tool is brought in for the first time. Using an example application, we examine the manual testing process, which is where the automation will start. The automation of test execution is normally accompanied by the automation of test verification, but even this leaves a lot to be desired in terms of a fully automated testing regime. This chapter shows why automating tests is not the same as automating testing.

2.1 An example application: Scribble

Our example application is called Scribble. It is a very simple word processor, for the purpose of illustrating the points we wish to make. Real applications will be far more complex, and that makes the problems of automating their testing correspondingly more complex as well.

2.1.1 Example test case: editing a List in Scribble

A feature of Scribble is that a document can include a structure known as a List. The example test case that we use in this chapter exercises some of the List editing functions. The way it works is illustrated in Figure 2.1.

Starting from the initial state with just one item in the List (Netherlands), we add a new item, Belgium (using the *Add Item* function), which is added to the end of the List. Next we sort the List, which changes

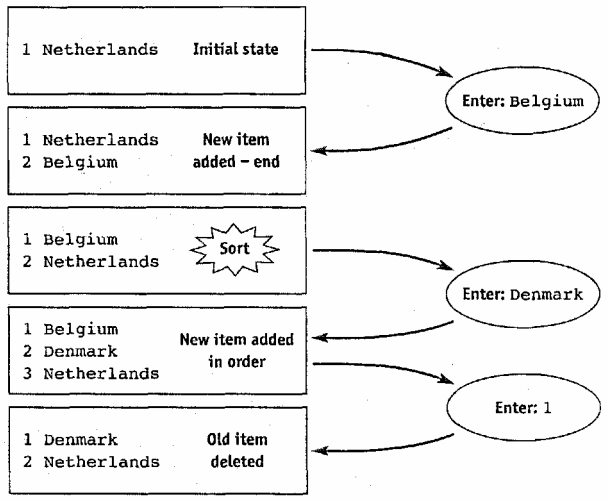


Figure 2.1 Examples of the List editing functions of Scribble.

the order of the two items. When we add the third item (Denmark), it is inserted into the List in the correct place, since it is now a sorted rather than an unsorted List. Finally, we delete the first item in the list (Belgium) by selecting *Delete item* and entering the position number.

2.1.2 The test for a Scribble List

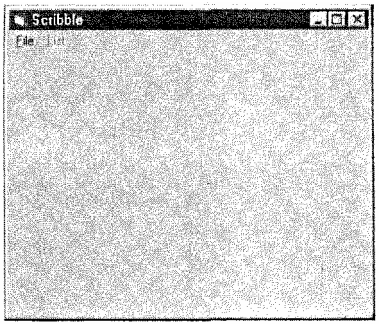
Now we are ready to test (manually) our application. We will test the editing of a List in a document called countries.dcm (the suffix 'dcm' stands for 'document'). The starting state is a sorted List of three items. The test case will perform the following:

- add two new items to the sorted List;
- move an item (which makes the List unsorted);
- add an item to the unsorted List;
- delete an item;
- try to delete an item which isn't there (invalid position number).

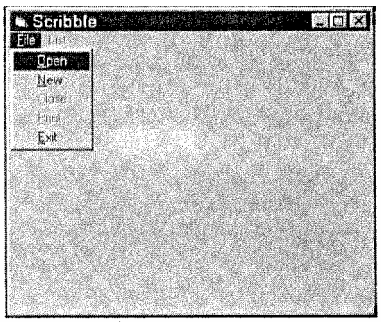
The edited document will be saved as countries2.dcm.

This is not a very rigorous test case and of course there will need to be others to adequately test this functionality. However, we have chosen the above test case as being reasonable for our purposes.

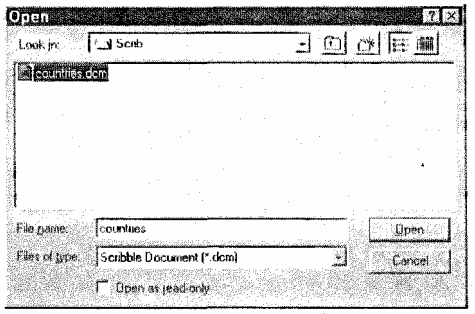
The execution of the test case is shown in the series of screen shots in Figure 2.2.



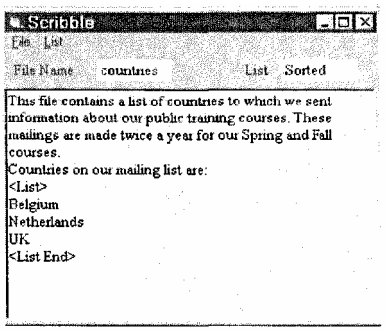
Screen 1 Invoke Scribble.



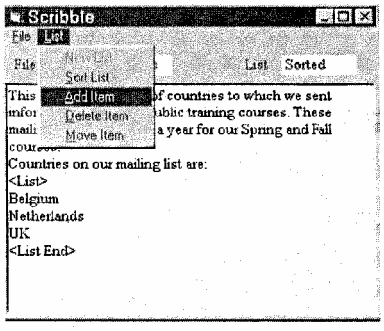
Screen 2 File menu: Open.



Screen 3 Select countries. dcm to open.

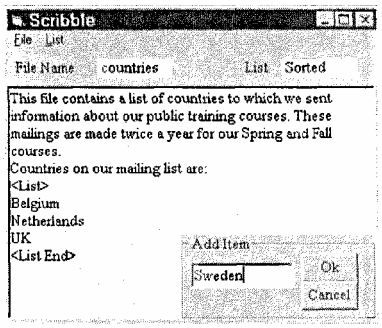


Screen 4 Countries file opened.

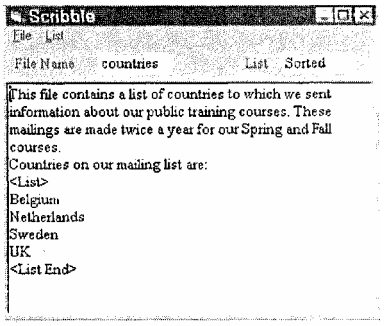


Screen 5 List menu: Add Item selected.

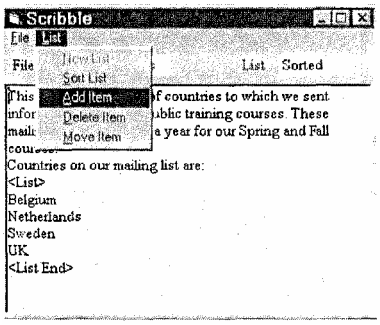
Figure 2.2 The screen shots for our example test case for Scribble.



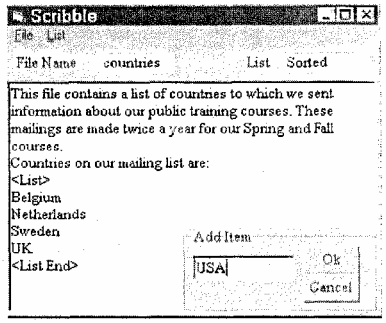
Screen 6 Add Item: Sweden.



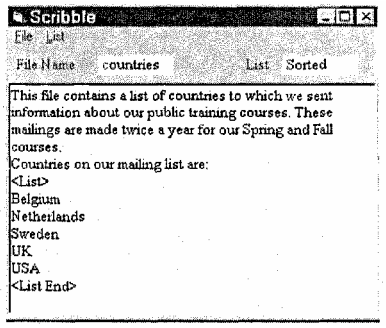
Screen 7 Sweden added in order.



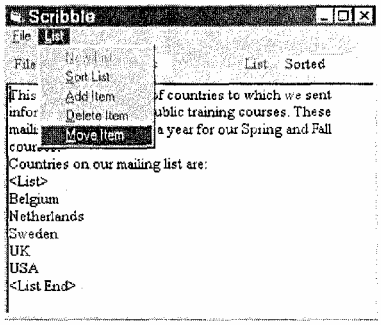
Screen 8 List menu: Add Item.



Screen 9 Add Item: USA.

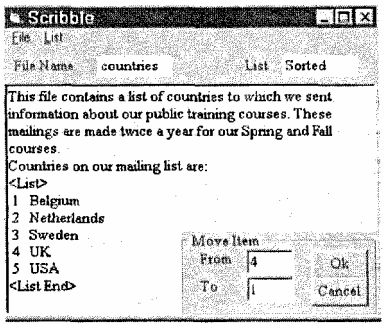


Screen 10 USA added in order.

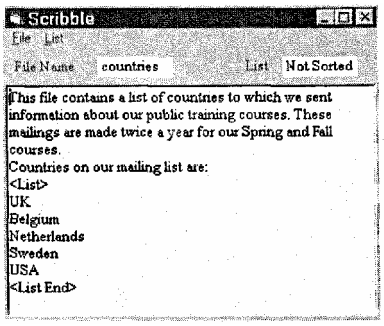


Screen 11 List menu: Move Item.

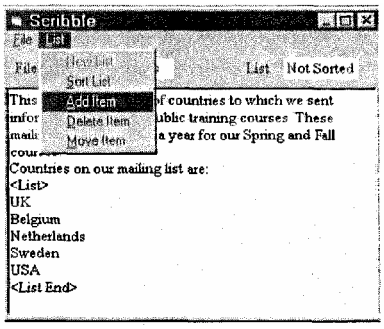
Figure 2.2 Continued



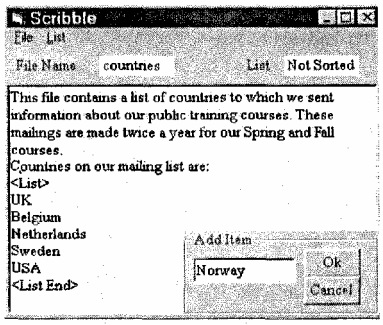
Screen 12 Move from position 4 to i.



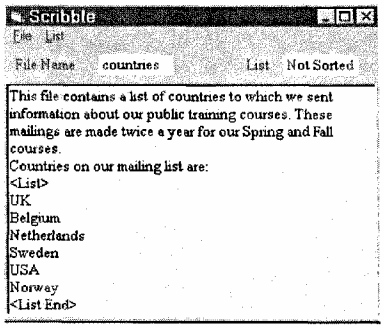
Screen 13 UK now at top (list unsorted).



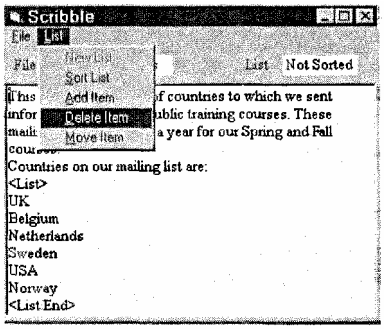
Screen 14 List menu: Add Item.



Screen 15 Add Norway to unsorted list.

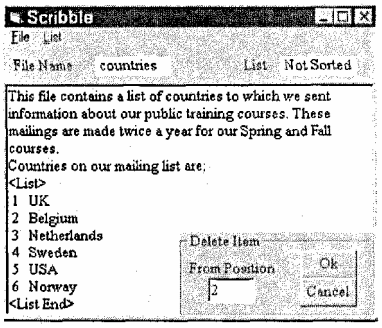


Screen 16 Norway added at end.

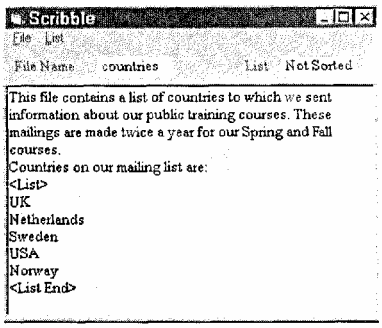


Screen 17 List menu: Delete Item.

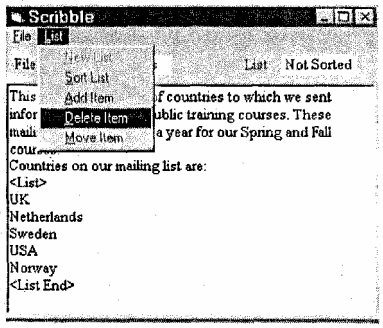
Figure 2.2 Continued



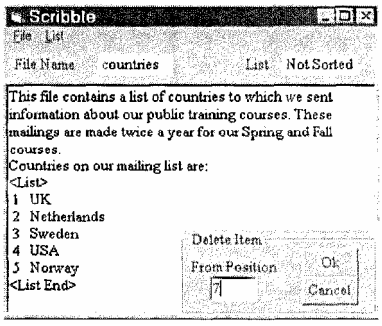
Screen 18 Delete item at position 2.



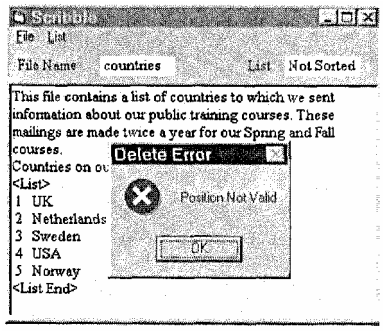
Screen 19 Belgium deleted.



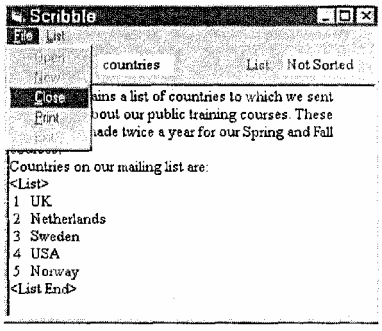
Screen 20 List menu: Delete Item.



Screen 21 Try to delete position 7.

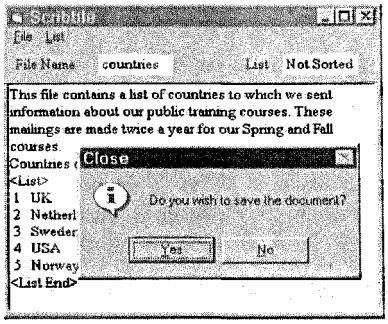


Screen 22 Error message: invalid position.

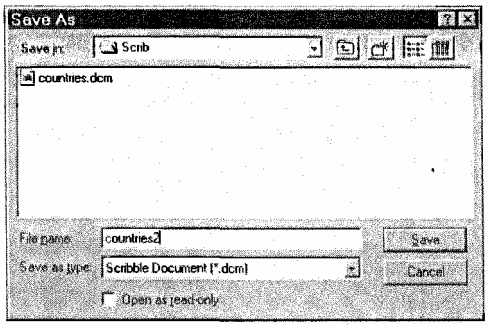


Screen 23 File menu: Close.

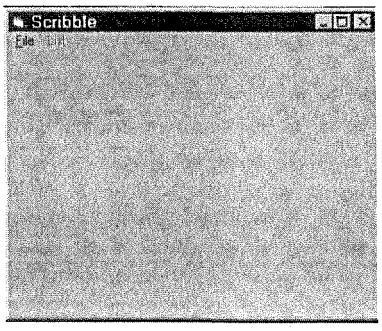
Figure 2.2 Continued



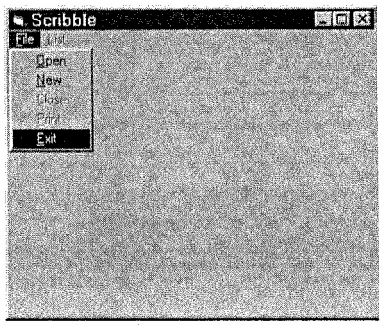
Screen 24 Wish to save?



Screen 25 Save as countries2.



Screen 26 Scribble main menu.



Screen 27 File menu: Exit.

Figure 2.2 Continued

2.1.3 What was the test input?

In the test case above, we have entered input to take the application from its initial invocation to completion, entering specific data to perform the test case. The specific key presses and/or mouse actions that we have performed are shown in Table 2.1. Note that we have described our actions in terms of the logical elements on the screen rather than their physical locations, for example, 'List menu' rather than 'bit position (445, 120).' Note also that all mouse clicks listed are left mouse button.

Table 2.1 What was actually entered to run the Scribble test.

Action/input

1 Move mouse to Scribble icon
2 Double click
3 Move mouse to *File* menu
4 Click
5 Move mouse to *Open* option
6 Click
7 Move mouse to countries.dcm in file list
8 Double click
9 Move mouse to *List* menu
10 Click
11 Move mouse to *Add Item*
12 Click
13 Type 'Sweden'
14 Move mouse to *OK* button
15 Click
16 Move mouse to *List* menu
17 Click
18 Move mouse *to Add Item*
19 Click
20 Type 'USA'
21 Move mouse to *OK* button
22 Click
23 Move mouse to *List* menu
24 Click
25 Move mouse to *Move Item*
26 Click
27 Type '4'
28 Press TAB key
29 Type T
30 Press Return key
31 Move mouse to *List* menu
32 Click
33 Move mouse to *Add Item*
34 Click
35 Type 'Norway'
36 Move mouse to *OK* button
37 Click
38 Move mouse to *List* menu
39 Click
40 Move mouse to *Delete Item*
41 Click
42 Type '2'
43 Move mouse to *OK* button
44 Click
45 Move mouse to *List* menu
46 Click
47 Move mouse to *Delete Item*
48 Click
49 Type 7'
50 Move mouse to *OK* button
51 Click
52 Move mouse to *OK* button
53 Click
54 Move mouse to *File* menu
55 Click
56 Move mouse to *Close*
57 Click
58 Move mouse to *Yes* button
59 Click
60 Type 'countries2'
61 Press Return key
62 Move mouse to *File* menu
63 Click
64 Move mouse to *Exit*
65 Click

2,2 The manual test process: what is to be automated?

The test case that we have just carried out manually in the example above is the test case that we will now try to automate using a **capture replay** tool. The amount of effort it takes to automate a single test case depends on a number of things and varies widely. Experienced automators typically report it takes between 2 and 10 times the effort required to run the test manually, and occasional reports have said it is as high as 30 times the manual test effort. Thus, a test case that takes half an hour to perform manually may well take between 1 and 5 hours to automate, but could also take as much as 15 hours. When you start automating, expect it to take at least five times longer to automate a test than to run it manually.

Some of the things that affect the amount of effort required to automate a test case include the following:

» The tool used. Tools provide a variety of facilities and features designed to make creating automated tests easy, and increasingly allow testers to specify during recording more about how the test should behave when being replayed.

« The approach to test automation. There are a number of different approaches we can use to create Automated tests, though the first approach most people use is to record the test being performed manually and then replay this recording. Other approaches require additional effort to modify recorded scripts or prepare scripts manually. While this does take longer initially the idea is to make it easier to implement new tests and reduce maintenance costs.

» The test automator's level of experience. Clearly, people who are very familiar with using a testing tool will be able to use it more quickly and with fewer mistakes. They should also be able to avoid implementations that do not work well and concentrate on those that do.

» The environment. Where the software to be tested runs in specific environments that are difficult to replicate precisely for testing, it can be more difficult to automate testing. For example, embedded software and real time software are particularly troublesome as these can require specialized tools or features that may not be commercially available.

» The software under test. The tests for software applications that have no user interaction, such as batch programs, are much easier to automate providing the environment in which they execute can be reproduced. This is because user interaction has to be 'programmed' and this can be a lot of work. Although this can be recorded it does not result in a cost-effective solution in the long term. A batch program has no user interaction but is controlled from information passed to it when invoked or read from files or databases during execution.

There arc also many facets of software applications that can make testing, and in particular automated testing, more difficult. There are many testing issues to consider when designing software applications

and often it is the design of the software itself that will make or break automating testing.

- The existing test process. When automating testing we are automating some of the five activities that make up the test process (identification of test conditions, design of test cases, build the tests, execution, and comparison of expected outcomes with actual outcomes) as described in Chapter 1. The consequences of this are our next point of discussion.

The effect of the existing test process on the effort required to automate testing is immense. If test cases are not documented but the testers make them up as they go, then automating the test cases will involve designing them as well. At the other extreme, if the test cases are documented in great detail such that every input and comparison is noted, then automating them will require less effort.

In the following subsections we describe three broad categories of test process: ad hoc testing, where the test cases have not been designed or documented, vague manual scripts, where the test cases are documented but not all the details have been specified, and detailed manual scripts, where every input and comparison is noted. Your own test processes should fit, more or less, into one or more of these categories.

2.2.1 Ad hoc testing: unscripted

Ad hoc or unscripted testing implies simply sitting down at the computer and trying things. The tester may or may not have a mental plan or checklist of things to test, let alone any specific values to try. He or she thinks up what to test, trying this and that, and thinking 'I wonder what will happen if I do this.' The tester's ideas and actions are probably not logged or documented, so what is done cannot be accurately repeated.

This is usually the case when a software development project is running late and little or no thought has been given to testing. Typically there are no specifications, the requirements are still changing, and the manager is saying 'there is no time to document, just test.' This is not a good place to be, but if that is where you are now, you have no choice about it. You have our sympathy, but don't try to automate this kind of testing - it will only slow you down even more.

The way in which automation of ad hoc testing often proceeds is described in this section, and illustrated in Figure 2.3.

The steps of unscripted testing, illustrated in Figure 2.3, are:

1. think what to do, what to test;
2. think up specific inputs;
3. enter the inputs that have just been thought up;
4. check that the software worked correctly, by watching the responses that appear on the screen.

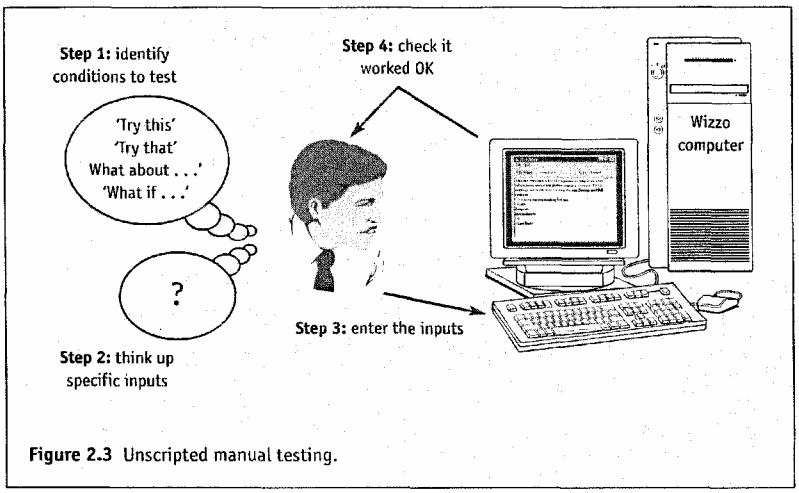


Figure 2.3 Unscripted manual testing.

It is hard to think of many advantages for unscripted testing. The one aspect that is most often mentioned is that it should save time, since we do not need to spend any time in planning and test design activities, we can just start testing. This is analogous to the 'why isn't Johnny coding yet?' syndrome that characterizes unsophisticated software development. This approach always costs more in the long term (both in writing software and in testing it) as more mistakes will be made and they will cost more to correct when they are found.

The disadvantages of unscripted ad hoc testing include:

» important areas that should be tested may be overlooked;

» other areas may be tested more than is necessary;

« tests are not repeatable, so fixes cannot be confirmed with certainty (in some cases defects may not be reproducible);

» it is usually both ineffective and inefficient.

Automating ad hoc testing relies on the automator deciding what is to be tested. This means that the tester must have the necessary skills to automate tests or the automator must have the necessary skills to define good test cases. Either way, it relies on individuals designing and implementing tests without any independent assessment of the quality of the tests.

2.2,2 Vague manual scripts

Vague manual scripts contain descriptions of the test cases without going into the detail of specific inputs and comparisons. For example, a vague manual script might say 'try some invalid inputs' or 'add some items to

the list.' The test conditions may be implied (some invalid inputs), rather than stated explicitly (for example invalid inputs that are too large, too small, or the wrong format).

Our description of the example test case is also an example of a vague manual script:

- add two new items to the sorted List;
- move an item (which makes the List unsorted);
- add an item to the unsorted List;
- delete an item;
- try to delete an item which isn't there (invalid position number).

This type of manual script usually includes a vague description of the expected results and a place to tick to indicate when a test has passed (to be filled in each time the test case is run). Therefore, the test script for our test case may look something like that shown in Table 2.2.

Table 2.2 Example of a vague manual test script based on our test case for Scribble.

Step	*Input*	*Expected result*	*Pass*
1	Run up Scribble	*File* menu displayed	File
2	Open file with sorted list	contents displayed	Items
3	Add some items to List	added in order	Item moved,
4	Move an item	List now unsorted	Item
5	Add an item	added at end of List	Item
6	Delete item from List	deleted	
7 to 8	Use invalid position number delete an item Save changes in new file	Error message displayed End of test	

The steps of manual testing from a vague script, illustrated in Figure 2.4, are:

1. read what to do;
2. think up specific inputs;
3. enter the inputs that have just been thought up;
4. check that the software worked correctly, by watching the responses that appear on the screen.

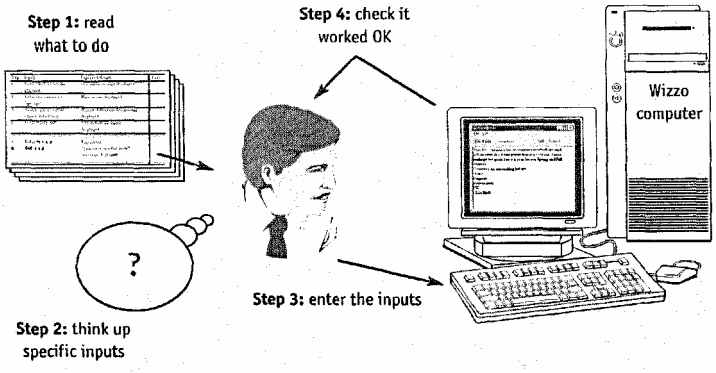


Figure 2.4 Scripted (vague) manual testing.

Vague manual scripts have many advantages over unscripted ad hoc testing:

- different testers have a reasonable 6hance of finding similar defects following the same script;
- the conditions identified will all be tested if the script is followed; • the script documents what is to be tested; • the test cases can be reviewed or inspected;
- the testing performed is likely to be more effective and more efficient than ad hoc testing (depending on how well the test design process is carried out).

However, vague manual scripts also have a number of disadvantages:

- the test input is not precisely defined;
- different testers will perform slightly different tests using the same script;
- it may not be possible to reproduce problems reliably.

When a test is run from a vague script, the tester still has to decide exactly what input to use in order to test the identified conditions. For example, in our example test case there are many different ways to have an invalid input: a number could be out of range, either above or below, it could be alpha instead of numeric, it could be a negative number or a space.

Automating vague manual test scripts relies on the automator deciding what specific inputs to use and possibly being able to decide whether or not the actual results are correct. This means that the automator does not have to invent the test conditions but must have an adequate understanding of

the application and the business to insure that the implementation of the test cases does not denigrate their quality.

2.2.3 Detailed manual scripts

A detailed manual script contains exactly what will be input to the software under test and exactly what is expected as a test outcome for that input. All the tester has to do is what the script says. This is the level that is the closest to what the test tool does, so automation is easiest in many ways from this level. However, this is also the most tedious and boring for human testers; they have no scope for creativity. Using a vague manual script, at least they can choose what inputs to use for each test.

If the vague script said 'enter some invalid input' or, slightly less vague, 'enter an invalid position number' or 'enter a position number too large,' the detailed manual script would say 'enter invalid position number 7.' Here, 7 is invalid based on the context of the test.

Our detailed manual script might look like Table 2.3.

Table 2.3 Example of a detailed manual test script based on our test case for Scribble.

Step	*Input*	*Expected result*
	Click on Scribble icon	Scribble opened, *File* menu enabled
1 2	Move mouse to *File* menu, click	Options available: *Open, New, Exit*
3	Move mouse to *Open* option, click	Filenames displayed, including countries. dcm, filename prompt displayed
4	Move mouse to countries. dcm, click	Text plus List of three countries: Belgium, Netherlands, UK, *File* and *List* menus displayed
		Item added as number 3 in List
7 12	Enter 'Sweden' Enter *'T*	*Position not valid* message displayed

Detailed scripts come in many 'flavours.' The example shown in Table 2.3 is typical of a very thorough manual script that can be performed by staff with relatively little knowledge of the application or its business use. The script's expected results are shown here as exact (as the most detailed form of detailed script). In this case, if the software is relatively stable, the test script could actually show the screen shots as expected results. The detailed script would then look like Figure 2.2. A slightly less detailed format for the expected results might say 'error message displayed' instead of giving the exact text of the expected error message, for example.

The steps of manual testing with a detailed script, illustrated in Figure 2.5, are:

1. read exactly what to do;
2. enter the exact inputs in the script;
3. check that the software worked correctly, by watching the response that appears on the screen.

The advantages of detailed manual scripts are:

» testers will run exactly the same tests (i.e. as exact as manual regression testing can be) from the same scripts (subject to human error, of course - which is not insignificant when people are bored);

» software failures revealed by the tests are much more likely to be reproducible;

* anyone can run the test, since the instructions are full and detailed;

» it is easier to automate since all the information about inputs and expected outcomes is given in the script.

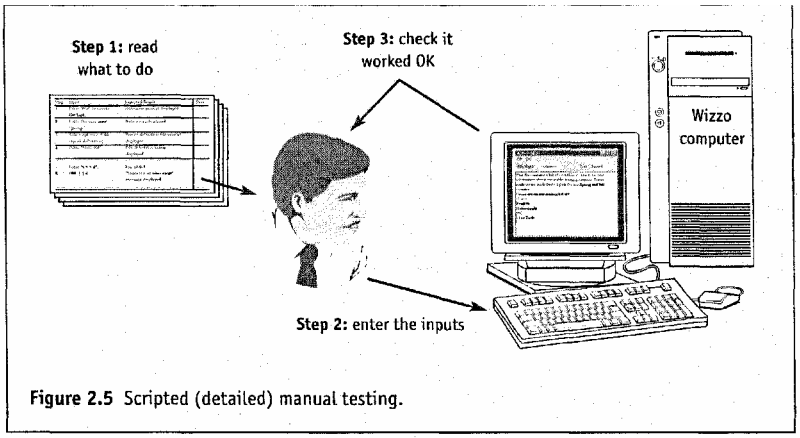


Figure 2.5 Scripted (detailed) manual testing.

The disadvantages of detailed manual scripts are:

- they are expensive to create (requiring a lot of effort);
- there is often a lot of redundant text (e.g. the word 'Enter' appears many times) though strictly there need not be so;
- scripts tend not to lend themselves to reuse of common actions;
- they are verbose, making it hard to find information and hard to maintain;
- they are very boring to run, since the tester has no scope to think or be creative, therefore mistakes are more likely.

Because a detailed test script contains detailed information about how to perform each test case, the tester does not need any knowledge of the application. Test tools have no knowledge of the applications they test; in fact test tools have no intelligence at all. They are, however, superb workhorses. They never complain, never become bored or distracted, and never want a holiday or sick leave, but will work all day and all night throughout the year. Unfortunately, they do need to be told in painstaking detail everything that they are required to do.

EXPERIENCE REPORT

In a conference presentation, the test tool was likened to a robot dog (called Squat). A dog is not as intelligent as a human tester, and robots are even less so. Some of Squat's characteristics are:

- he gets sick if he is not fed properly (i.e. good quality scripts, with error recovery built in, for example);
- he gets unfit if he is not exercised (if left for a long time without being used, he won't do very much when he is eventually woken up again);
- he gets bored if he is not taught new tricks (need to keep developing the automated regime over time).

Source: Paul Godsafe (Instem-Apoloco), 'Automated testing,' British Association of Research Quality Assurance 13th International Conference, Glasgow, May 1998.

A detailed script is, therefore, in theory a good starting point for automation because all of the details are specified. However, detailed manual scripts are not usually written in a form that any test tool can understand. There is then an overhead associated in translating the manual scripts into automated scripts that a test tool can process. The information is therefore documented twice, once in the manual scripts and again in the automated scripts. Although unavoidable when automating existing detailed tests, this is a duplication that would be wasteful if new tests were documented in the same way.

For efficiency, a new form of test documentation is required, one that can be read and understood by humans and used directly by a test tool.

Automating detailed manual test scripts frees the automator from responsibility for the quality of test case design, allowing him or her to focus on how best to automate the tests. The automator does not have to understand the application, the business or the test cases because all the information is documented.

23 Automating test execution: inputs

It is natural to assume that automating test execution must involve the use of a test execution tool, but in fact this is not the only way to achieve at least some test execution automation. For example, batch programs and other programs that do not involve user interaction are perhaps the easiest to automate. A simple command file can be used to set up data, invoke the program, and compare the outcomes. Although it will lack most of the additional facilities offered by modern testing tools it will automate the essential chores. Other options may be applicable to those for whom a commercial tool does not exist, is not suitable, or is not available; for example, post-execution comparison, described in Chapter 4, pre- and post-processing, described in Chapter 6, or changing the application so it can run in 'test mode.'

2.3.1 Automating test input

In the rest of this section, we will assume that a commercial test execution tool will be used to automate testing, since this is what most people will do. We turn on the record or capture facility of our test tool while we perform all of the actions for the manual test case shown in Table 2.1.

The tool will record all of the actions made by the tester and will write them to a script in a form that the tool can read. Our recorded script will contain the test inputs (including text and mouse movements). The tool can then repeat the tester's actions by reading the script and carrying out the instructions in it. The script will be held in a file that can be accessed and edited.

Scripts can also be created manually, but because they are generally written in a formal language that the tool can understand, writing (or editing) scripts is best done by people with programming knowledge. This makes a test tool's capture facilities appear very attractive indeed since the script generated during the recording of a test case performed manually is the same script needed by the tool to perform exactly the same test case.

2.3.2 Replaying the recorded script

Let us assume that we have recorded our example test case being performed manually and now wish to replay it. Figure 2.6 depicts what will happen. The test script (that we have named countries.scp) is read by the tool, which then invokes Scribble and drives it as though it were a real person at the computer keyboard. The actions the test tool takes and the information it effectively types into Scribble are exactly as described in the test script. During execution of the test case, Scribble is caused to read the initial document countries.dcm, output information to the computer screen, and finally output the edited document as a file called countries2.dcm.

The test tool creates a log file countries.log containing information about what happened as the test case was run. Logs usually include details such as the time it was run, who ran it, the results of any comparisons performed, and any messages that the test tool was required to output by instructions within the scripts.

2.3.3 Automated scripts are not Like manual scripts

Exactly how the script looks will depend on the test tool being used and the implementation of the application.

Table 2.4 gives an idea of what a recorded script may look like for our Scribble test. What is listed here is a pseudo-code version of the minimum information that a tool would produce, but this should give you an idea of the complexity of a script for a rather simple test.

The scripting languages for the commercial tools do basically the same thing, but often in different ways. To give you an idea of what some scripting languages look like, Figure 2.7 shows a portion of the Scribble test in a commercial scripting language. Actually, this may not be exactly correct, as

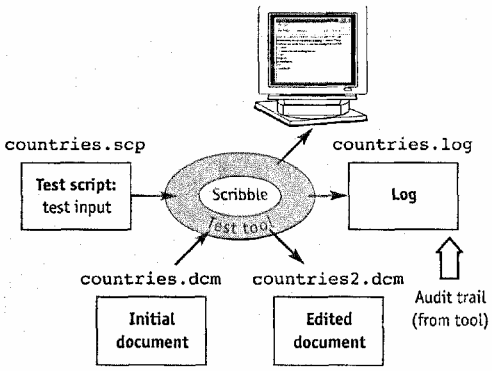


Figure 2.6 Using a test tool to replay a script that is the recording of our example test case on our word-processing application Scribble.

Table 2.4 Pseudo-code recorded script for the Scribble test.

```
LeftMouseClick 'Scribble'
FocusOn 'Scribble'
SelectOption 'File/Open'
FocusOn 'Open'
Type 'countries'
LeftMouseClick 'Open'
FocusOn 'Scribble'
SelectOption 'List/Add Item'
FocusOn 'Add Item'
Type 'Sweden'
LeftMouseClick 'OK'
FocusOn 'Scribble'
SelectOption 'List/Add Item'
FocusOn 'Add Item'
Type 'USA'
LeftMouseClick 'OK'
FocusOn 'Scribble'
SelectOption 'List/Move Item'
FocusOn 'Move Item'
Type '4'
Type <TAB>
Type '!'
LeftMouseClick 'OK'
FocusOn 'Scribble'
SelectOption 'List/Add Item' FocusOn 'Add Item'
Type 'Norway' LeftMouseClick 'OK' FocusOn 'Scribble'
SelectOption 'List/Delete Item'
FocusOn 'Delete Item'
Type '2'
LeftMouseClick 'OK'
FocusOn 'Scribble'
SelectOption 'List/Delete Item'
FocusOn 'Delete Item'
Type '7'
LeftMouseClick 'OK' FocusOn 'Delete Error' LeftMouseClick 'OK' FocusOn 'Scribble' '
SelectOption 'File/Close'
FocusOn 'Close'
LeftMouseClick 'Yes'
FocusOn 'Save As' Type 'countries2' LeftMouseClick 'Save' FocusOn 'Scribble'
SelectOption 'File/Exit'
```

the example has been constructed rather than recorded. Also, the tools may have changed their scripting languages since this book was written. However, it does give an idea of what a script might actually look like.

Some scripts may look quite different, but regardless of what our script language looks like, each is automating only one thing: the actions performed.

Depending on the format of the recorded test, there may also be a great deal of redundant information contained in the captured script, such as the pauses between entering data (while reading the manual script, for example), and any typing mistakes that were corrected (backspacing over incorrect characters before typing the correct ones).

In order to make a recorded script usable, or indeed even recognizable, comments should be inserted in the script. Despite some vendor claims, no recorded scripts 'document themselves.'[1] We expand on these matters in Chapter 3.

[1] We feel that the comments produced in 'self-documenting' scripts are about as useful as an ashtray on a motorbike.

```
;       Select "&List~&Add Item" from the Menu
        MenuSelect "&List~&Add Item"
        Pause 3 seconds

; Use the dialog window
Attach "Add Item ChildWindow~1"
        Type "SWE{Backspace}{Backspace}"
        Pause 2 seconds
        Type "weden"
        Button "OK" SingleClick

; Use the parent window
Attach "Scribble MainWindow"

;       Select "&List~&Add Item" from the Menu
        MenuSelect "&List~&Add Item"
        Pause 2 seconds

; Use the dialog window
Attach "Add Item ChildWindow~1"
        Type "USA"
        Pause 1 seconds
        Button "OK" SingleClick
```

Figure 2.7 Script segment from a commercial testing tool.

All scripting languages allow you to insert comments into the scripts. However, just as when commenting code, you must conform to the syntax or the test tool will try to use your comment as a test input (just like a compiler will try to compile your comment in source code if it does not conform to comment syntax).

2.3.4 Benefits and uses of automating only test input

A raw recorded script (of what a tester did) does have its place within a test automation regime, but it is a much more minor place than most novices to test automation give it.

One advantage of recording manual tests is that you end up with a form of replayable test comparatively quickly. Little or no planning is required and the tool can be seen to provide benefit more or less right away (well, at least in theory). However, recording ad hoc testing is a bit like writing software without a specification. You may end up with something similar to what is actually required but is unlikely to be any better than unspecified software. The end product will probably do some things correctly, some things incorrectly, not do some things, and do others that are not required.

One benefit of capturing and recording test inputs is that this can automatically provide documentation of what tests were executed. This provides an audit trail, which is useful if you need to know exactly what was done. This audit trail does not necessarily make a good basis for an effective and

efficient automated testing regime, but it has its place as a way of getting started and for special tasks. For example, if you have invited users in for 'playtime' and they manage to crash the system, you can at least see what was typed in. The audit trail may also contain a lot of useful information that is not normally available from manual tests, for example detailed statistics on timing or performance.

The set-up of test data is often a long and tedious task and for this reason is sometimes avoided, resulting in fewer runs of a set of test cases than there should be. If the data set-up can be recorded the first time it is performed then, providing the software's user interface does not change, the set-up can be replayed much more reliably and usually a lot faster by the tool. For example, adding the details of one new client isn't too bad, but adding the details of 100 new clients is best automated. Even if the next version of software has user interface changes that invalidate the recording, it may still be worth doing, since the additional effort to record the data set-up the first time will soon be repaid by the effort saved with the first or second replay. This is only true if the recording does not need editing; where editing is necessary the costs are much greater.

There are some other uses of recording manual tests, or at least recording of user interaction, such as making the same maintenance change across large numbers of data files or database(s). A recording can also serve as a demonstration. For multi-platform applications a script may be able to be recorded on one platform and played back on a number of other platforms with no or only minor changes. If complicated sequences need to be typed accurately, and this is error-prone if done manually, replaying them as one step in a manual test can save time. Automated execution is much more accurate than manual execution, and therefore the test input is repeatablc. Exact reproduction of inputs is particularly useful in reproducing bugs. If the cause of the bug has been recorded, playing it back should reproduce it, and playing it back after it has been fixed should confirm that it was fixed correctly (or not).

However, in most circumstances, automating only test execution has little benefit. Manual verification is error prone, wasteful, and very tedious. Yet some people end up with a test execution tool which produces stacks of paper that has to be checked by hand - this is nonsense if we are talking about automating testing. In order to claim any real test automation, we should therefore also automate comparison of the test outcomes.

It is important to appreciate that we are not suggesting that test result comparison for all test cases should be automated. As we have already discussed in Chapter 1, not all test cases should be automated, and not all comparisons should be automated either.

2.3.5 Disadvantages of recording manual tests

The disadvantages of record and playback only become apparent as you begin to use the tool over time. Capture replay always looks very impressive when first demonstrated, but is not a good basis for a productive long-term test automation regime.

The script, as recorded, may not be very readable to someone picking it up afterwards. The only value of an automated test is in its reuse. A raw recorded script explains nothing about what is being tested or what the purpose of the test is. Such commentary has to be inserted by the tester, either as the recording is made (not all tools allow this) or by editing the script after recording has finished. Without this information any maintenance task is likely to be difficult at best.

A raw recorded script is also tied very tightly to the specifics of what was recorded. Depending on the tool, it may be bound to objects on the screen, specific character strings, or even screen bitmap positions. If the software changes - correction: *when* the software changes - the original script will no longer work correctly if anything to which it is tightly bound has changed. Often the effort involved in updating the script itself is much greater than re-recording the script while running the test manually again. This usually does not give any test automation benefit. For example, the values of the inputs recorded are the exact and detailed values as entered, but these are now 'hard-coded' into the script.

The recording is simply of actions and test inputs. But usually the reason for running a test is to look at the test outcome to see whether the application does the right thing. Simply recording test inputs does not include any verification of the results. Verification must be added in to any recorded script in order to make it a test. Depending on the tool, it may be possible to do this additional work during recording (but it takes additional time and effort) or during the first replay, otherwise it will be necessary to edit the recorded script. This is described in more detail in Section 2.4.

2.3.6 Recommendation: don't automate testing by simply recording

It should be the aim of every professional testing organization to implement an efficient and effective test automation regime appropriate for their objectives. Simply recording manual tests will not achieve this for the following reasons:

- Unscripted (ad hoc) testing is not usually very effective (it can be effec tive in some cases if you have a person who either knows the system well and has experienced the weaknesses of previous versions, or has a good understanding of how the software works and is able to identify likely defective areas). Automating ad hoc testing will at best result in faster, but nonetheless not very effective, testing.
- Recording only input does not result in an automated test. The out comes of a test have to be checked to insure they are as expected; if this is not automated, this is not an automated test, only automated input.
- Re-recording when the software changed is inefficient. Almost any change to the user interface of software will result in changes to the test inputs or their order or both. Editing recorded scripts is neither easy nor pleasant. It is often easier and quicker to simply re-record the test but this usually incurs an unacceptably high cost anyway.

We recommend that if you currently perform only ad hoc manual testing, you will achieve far more benefit from improving your testing process than from attempting to automate it. The use of testing techniques, for example, may well double your test effectiveness. (See the threefold improvement in Chapter 12.) This will save you much more money than trying to run ineffective tests faster and will lead to better quality software.

2.3.7 Automated execution, with manual verification?

As it stands we have a script that was generated by recording our example test case being performed manually, and this can now be replayed by our test tool as often as we wish. However, all we have automated are the test inputs. In order to determine if the test case produced the correct results, we still have to check the expected outcomes manually. The tool will not do any checking for us yet because we have not instructed it to do so; so far, it knows only how to enter one set of test inputs, the inputs of our one test case.

There is a certain novelty value in having a tool type the inputs in for you, especially at first. There is a tendency for 'coffee parties' to gather around the screen. Remarks such as 'I wish I could type that fast' may be overheard. However, the novelty will soon wear off when the testers realize that they still have to sit there watching the test in order to verify that it worked (or not). The idea of overnight and weekend testing will not be popular with testers if automation has only reached this stage as yet.

Manually checking that our test case has worked correctly could involve having to watch the screen as the tool replays the script. It would also be a good idea to take a look at the edited document (countries2.dcm) after test execution to insure that the data was saved as it had appeared on the screen. (See the Experience Report in Section 2.5.2.)

So do we have automated testing? Well, we have some automation: the inputting has been automated. However, we still have very much a manual process in checking that the software did the right thing.

2,4 Automating test result comparison

It is unlikely that the entire outcome of a test case will be compared (including not only what has changed but also everything that has not changed), since it is often impractical and unnecessary. Usually, only the most important or most significant aspects need to be compared. Some outcomes such as the outputs to the screen can be performed while the test is being executed, this is known as **dynamic comparison.** Other outcomes such as some outputs to a file or database can only be compared after the test case execution has completed, this is known as **post-execution comparison.** An automated test may need to use a combination of dynamic and post-execution comparisons. This topic is discussed in more detail in Chapter 4.

There are a number of design decisions that need to be taken when deciding how to implement automated comparison. We will illustrate with an example.

2.4.1 Deciding when to compare the outcome of a test case

The first design decision is to decide when in a test run a comparison should be done. As an example, take a single input from our Scribble test, to add an item (Sweden) into a sorted List. Let us assume that we are recording the manual test once again but this time we will insert a few dynamic comparison instructions as we go. We arrived at the point in the test case where we are going to add the country 'Sweden' to the list. Figure 2.8 shows the screen as it might appear at this stage, before and after entering 'Sweden'.

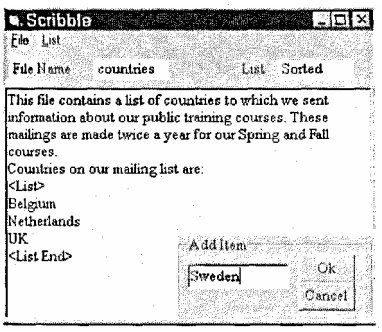
At this point, we should check the screen as it now appears (Screen 7) whenever this test is run in the future, so we need to add a comparison to the test script now.

The first step is to manually verify that this screen is indeed correct for the input of 'Sweden'. Next we will want to capture what is on the screen to compare against the next time this test is run. This will insure that the application performs in the same way at this stage of the test case each time it is run. The next step is to decide how much to compare.

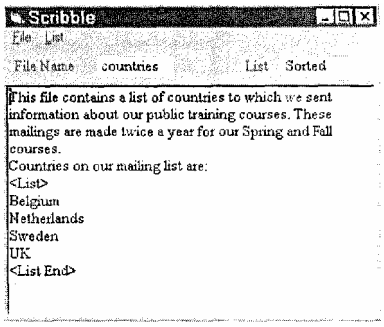
2.4.2 How much should be compared?

We now have some options for what to capture as the future expected output:

- the whole screen;
- the absolute minimum;
- something between these two extremes.



Screen 6 Add Item: Sweden.



Screen 7 Sweden added in order.

Figure 2.8 Screen images at the point when the list item 'Sweden' is added.

Capturing the whole screen is often the easiest to do. However, if there is any information on the screen which is likely to have changed the next time the test is run, then the easiest choice now will cause us significant grief later. Sometimes a screen will be displaying the current date or time. The next time the test case is run, we certainly do not want the test to fail because the test was not run on exactly the same date or at the same time it was recorded. In our example, we may have changed the text outside the List, but that is not important for this test, which is concerned only with whether an item can be added in the correct order to the List.

Capturing whole screens can be quite greedy of disk space for tests. Particularly when graphics are involved, the space needed to store a whole screen can run into tens of megabytes.

What about other options? We could perhaps capture only a single character, say the letter 'S.' Or we could capture the line containing the new entry, which would make sure that all of the new entry had gone in correctly. In Figure 2.9, we show this option as the shaded area.

Our first try at capturing an expected outcome seems sensible: it is a fairly minimal set. However, there is one error that may occur which this option will not find. Suppose the next version of the software ends up overwriting the third entry instead of moving UK down and inserting Sweden as the new third entry. The screen would then appear as shown in Figure 2.10.

The tool will pass this test, since all it is checking for is the line containing Sweden, and that looks exactly as it was told to expect it to look. So by making our comparison more robust (from things we are not interested in), it has become less sensitive to unexpected errors. (These terms are explained more fully in Chapter 4.)

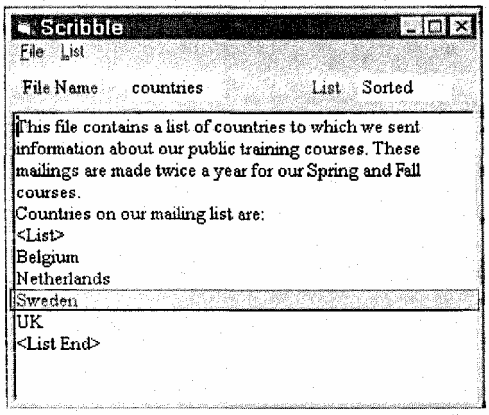


Figure 2.9 Possible selection for a comparison to verify whether 'Sweden' has been added correctly.

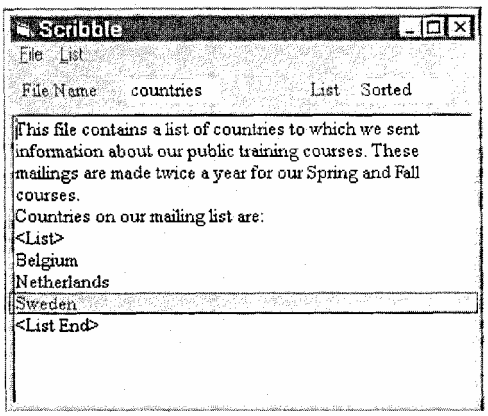


Figure 2.10 Screen image as it might appear if the application erroneously overwrote the third list entry with 'Sweden'.

A better option may be to compare not only the newly inserted entry, but also the entry that was 'pushed down.' This will catch the error shown (but may well miss still others). Our improved test comparison area is shown in Figure 2.11.

We have looked at the choice between comparing a little versus comparing a lot at a time. There are a number of other choices for comparison, which are discussed in more detail in Chapter 4.

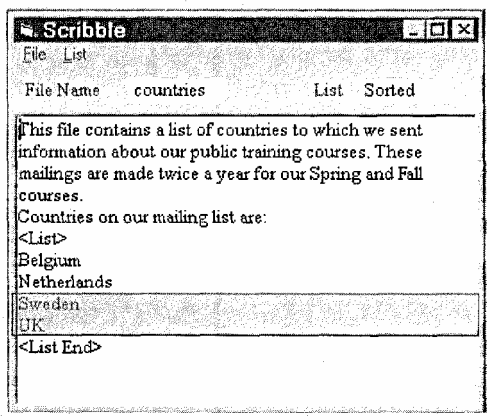


Figure 2.11 Improved selection for comparison.

2.4.3 Dynamic comparison

Let us now assume that we have included some dynamic comparison instructions as we recorded our example test case being performed manually, including the comparison for Sweden outlined above. Figure 2.12 depicts what will happen. The test script now contains additional instructions that tell the tool which outputs are to be compared and the expected result to which they arc to be compared. At each comparison point, the tool will capture the actual output from the software as it executes the tests and it will then compare that with the corresponding expected output. Every time a comparison is performed a message is written to the log file indicating whether or not differences were found. The test script may now look something like the example shown in Figure 2.13.

Note that even if a comparison reveals differences, the tool will continue to run the test case. This is both sensible and useful since not all differences imply catastrophic failure so it usually is worth running a test case right to the end to sec if any other problems can be revealed. However, this characteristic is something of a double-edged sword. If a detected difference really docs imply catastrophic failure, the tool will continue to run the test case anyway, unaware of the potential havoc doing so may cause. It may be that for a more complex system than our example word processor, test cases that do not go according to plan are best aborted to prevent undesirable outcomes such as data being incorrectly changed or deleted, large volumes of data being created or printed, or just hours of machine time being wasted.

Fortunately, where such potential damage could occur, it is usually possible to instruct the test tool to abort the test case. It will be necessary to edit the script though, inserting appropriate instructions to detect an undesirable situation (or perhaps more easily, not a specific desired situation)

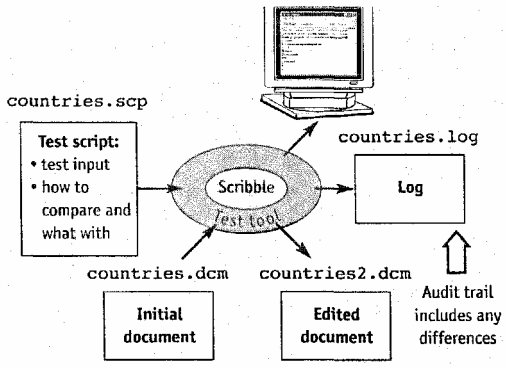


Figure 2.12 Using a test tool to replay a script that is the recording of our example test case on our word-processing application Scribble, but now including dynamic comparison instructions.

```
;                    Select    "&List~&Add    Item"
from the Menu MenuSelect    "&List~&Add    Item" Pause
3    seconds
;    Use the dialog window Attach    "Add    Item
ChildWindow-1 " Type    " SWE {Backspace } { Backspace } "
Pause    2    seconds Type    "weden" Button    "OK"
SingleClick
;    Use -the parent window Attach    "Scribble
MainWindow"
;                Text    check    on    Scribble Check ("
SCRIBBLE. CB",    "ADD_1"    )
;    Use the parent window Attach    "Scribble
MainWindow"
;                    Select    "&List~&Add    Item"
from the Menu MenuSelect    "&List-&Add    Item" Pause
2    seconds
;    Use    the    dialog window Attach    "Add Item
ChildWindow-1" Type    "USA" Pause    1    seconds
Button    "OK"    SingleClick
```

Fi(|ure 2.13 Script segment from a sample commercial testing tool showing a dynamic
co mparison instruction in bold. This instructs the test tool to perform a comparison
ac cording to instructions stored in the file SCRIBBLE. CB under the label ADD_1. These
in structions would have been previously set up in the tool's dynamic comparison database.

and then the action to take. This is 'programming' the script, and brings with it not only endless possibilities of tailoring test cases to detect all sorts of situations and deal with them accordingly, but also unconsciously to make errors that will cause the test case to fail through no fault of the software under test. This can lead to much wasted effort, instead of the intended saved effort.

2.4.4 Post-execution comparison

In the previous subsection we looked at how the use of dynamic comparison affects our example test case. Here we will look at the effect post-execution comparison would have on it as an alternative to dynamic comparison. A comparator tool for post-execution comparison is sometimes supplied bundled with a capture/replay tool and in any case most computer systems offer some utility program or other that can perform simple independent comparisons.

Figure 2.14 depicts what happens with our example test case when we use post-execution comparison. The test script is back to its simplest form: a straight recording of the inputs. The log file contains just the basic information including start and end times, who ran it, etc. but no information about the results of comparisons (though this will depend on the implementation). The expected outcomes are now held in a separate file called countries2.dcm, the same name as the output file with which it has to be compared, but in a different directory, rather than being embedded in the script. We could have used a naming convention such as Expcountries2.dcm to distinguish our expected results file from the actual results file. Using the same name is what we recommend, for reasons outlined in Chapter 5. After the test case has been executed the comparison process can begin. This will take the actual outcome of the test case and compare it with the expected outcome, possibly generating another log file (not shown in the figure) and/or a report of the differences found.

Here we have shown the edited output file (countries2.dcm) as the only outcome being compared. Whether or not this is sufficient depends on the purpose of the test case. This may be all that we consider necessary to compare, as it insures that the edits to the input file are reflected correctly in the new version of the file. While this may be a reasonable approach for positive testing, i.e. testing valid inputs such as adding Sweden correctly, it would not be appropriate for testing invalid inputs such as trying to enter an invalid position number. Showing that the screen display is updated correctly is more difficult to do in post-execution comparison, but it is not impossible. For these kinds of test, other outcomes would need to be captured in another file or files for later comparison.

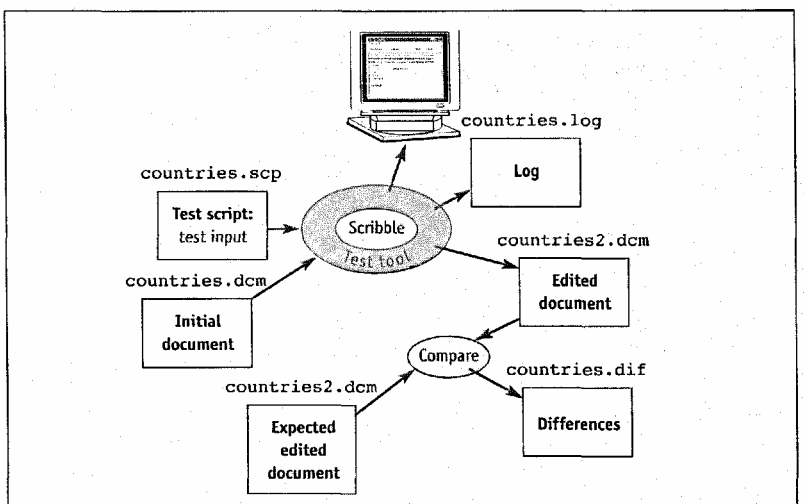


Figure 2.14 Using a test tool to replay a script that is the recording of our example test case on our word-processing application Scribble, with post-execution comparison.

Note that the diagram in Figure 2.14 implies two separate processes; the first executes the test case while the second process performs a comparison. It is this aspect that makes post-execution comparison a more difficult facility to automate well (Chapter 4 shows how to overcome these problems). Commercial testing tools usually provide much better support for dynamic comparison than they do for post-execution comparison.

2.4.5 Automated comparison messages have to be manually checked

So far we have described the automation of our example test case starting with a simple recording and then adding dynamic comparison and post-execution comparison to verify the actual outcome. Now, in order to determine whether or not our test case is successful every time it is run, there are some additional tasks we may have to do, depending on the tool and the implementation of post-execution comparison when used.

First we should look at the log file to make sure that the test case was executed right through to the end. It is quite possible for automated test cases to fail to start or be aborted part way through (for example, due to insufficient disk space). Where dynamic comparison has been used we may also need to check the log file for any 'failed' comparison messages. Where we have used post-execution comparison, we may have to look at one or more other files for information about any differences between the actual and expected outcomes.

Clearly, this is a manually intensive task. Having to repeat it for only a few test cases will not be too arduous, but it will be impractical where there are tens, hundreds, or thousands of test cases. Of course, it may be a relatively simple matter to write a small utility program to search through all the log files, doing a text match on the word 'fail,' wouldn't it? That utility program would need to know the names of all the log files, and where to find them. It may also give some spurious results, such as a test that writes text to the log file containing the word 'fail,' e.g. 'This is test 325B.43 Communication Link Failure Test.'

This additional utility should be produced if we want testing to be automated, but this does represent an additional task that is normally not anticipated.

At the time of writing, some commercial test tools do provide an acceptable summary of the information, or at least can be made to do so. However, where post-execution comparison has not been integrated well with the execution of the test cases, obtaining a report that takes into account the results of post-execution comparisons can be far more difficult. When we invoke a set of automated test cases we would really like to be presented with a single report at the end that takes into account all of the comparisons and simply states how many test cases ran successfully, how many failed, and then which test cases failed. (Our recommendations for the content of such a report are discussed in Chapter 9.)

2.4.6 Automating test comparison is non-trivial

2.4.6.1 There are many choices to be made

There are a number of aspects to automating comparison that must be designed into the automated testing regime. Whether and when to do dynamic comparison and post-execution comparison is one example. It is not a case of one alternative being better than another; each alternative has advantages and disadvantages, and each will be more suitable in some circumstances than in others. Usually a combination of the two should be used to give the best results.

Other choices include comparing a lot of information at once versus comparing only a little piece of information each time, and performing many comparisons versus performing only a few. (These and other issues are discussed further in Chapter 4.)

Many of the choices come down to trading off the resilience of the tests to changes in the software, with the ease of finding defects when the tests find a discrepancy.

2.4.6.2 Scripts can soon become very complex

Incorporating comparison into test cases is essential in order to be able to fully automate testing. If dynamic comparison is embedded in the script, as we have seen in the example, this can make the script more complex, since the comparison instructions and possibly the expected output will now be contained within the script.

The more information the script contains, the more susceptible it will be to changes, cither changes in the software being tested, or changes in the test cases themselves. For example, if the error messages changed their wording slightly, all expected outputs embedded in scripts would need to be edited. This can increase the cost of maintaining the scripts considerably. Ways to reduce such script maintenance costs are addressed in Chapter 3.

2.4.6.3 *There is a lot of work involved*

Tool support is essential, not only to automate testing, but also to help automate the automation! In order to produce an effective and efficient automated testing regime, effort needs to be invested upfront, so that effort is continually saved later on. The speed and accuracy of *use* of the testing tools is important. The effort involved in setting up this regime will be paid back many times on all of the numerous occasions when the automated tests arc re-run. The more reuse of automated tests you need, the more important it is to build them correctly.

2.5 The next steps in evolving test automation

Surely having put all this effort into automating test execution and comparison, we must be ready to tell the test tool to run our test cases so we can then put our feet up, right? Unfortunately, we have only just begun. What happens next? Typically, what is shown in Figure 2.15.

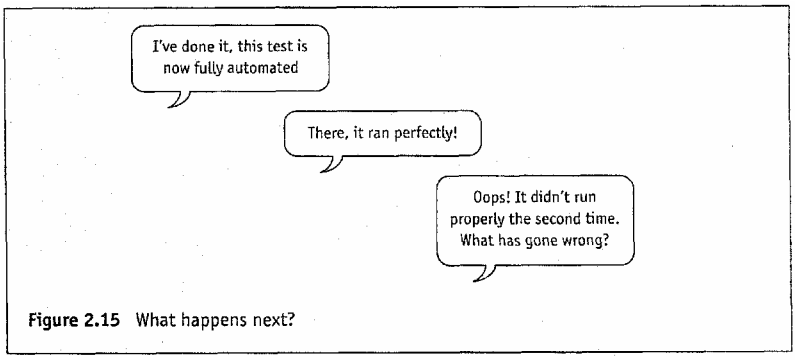
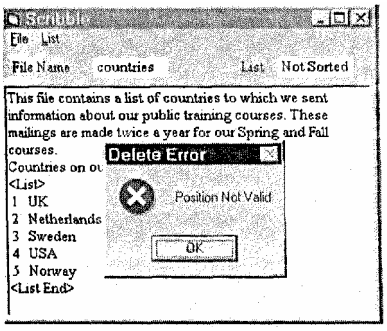


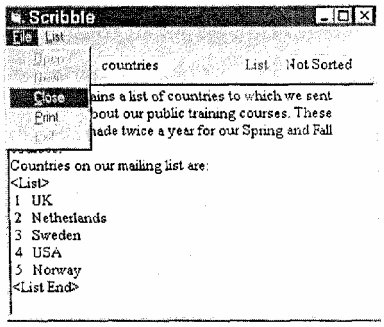
Figure 2.15 What happens next?

2.5.1 Why tests fail the second time (environmental trip-ups)

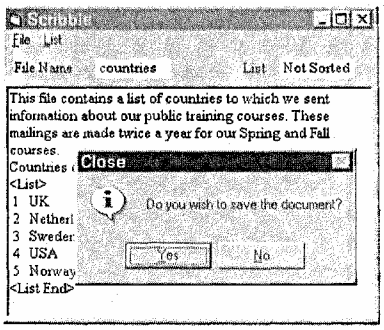
The first time our test ran, it used as its input the file countries.dcm. It created the file countries2.dcm as the edited document file. This is no problem when running the test for the first time, either manually or automatically. The final steps of our test script are shown in Figure 2.16.



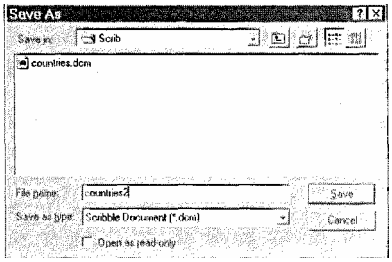
Screen 22 Error message: invalid position.



Screen 23 File menu: Close.

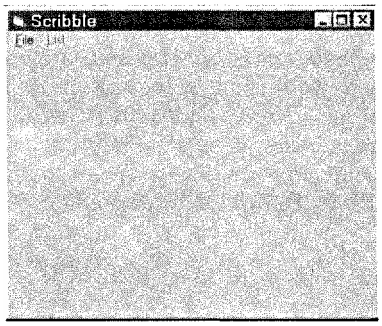


Screen 24 Wish to save?

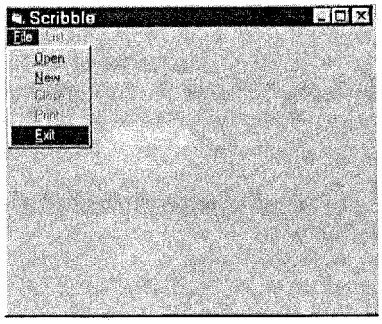


Screen 25 Save as countries2.

Figure 2.16 The end of our original test.



Screen 26 Scribble main menu.



Screen 27 File menu: Exit.

Figure 2.16 Continued

What happens the second time we run this exact test? Screen 25 will actually be different, because both countries.dcm and countrics2.dcm will be shown in the file list. The tool, however, will not 'notice' this! The next input is to type in countrics2. The next screen that appears is now not the one shown on Screen 26, but a new one containing a dialog box that was not seen the first time the test was run. This is shown in Figure 2.17.

If this happens when you arc testing manually, you would naturally put in the additional step of answering 'Yes' (and you would also probably have noticed the file in the list on Screen 25), and then select *Exit* from the *File* menu. You may not even realize that you had put an extra step into the test.

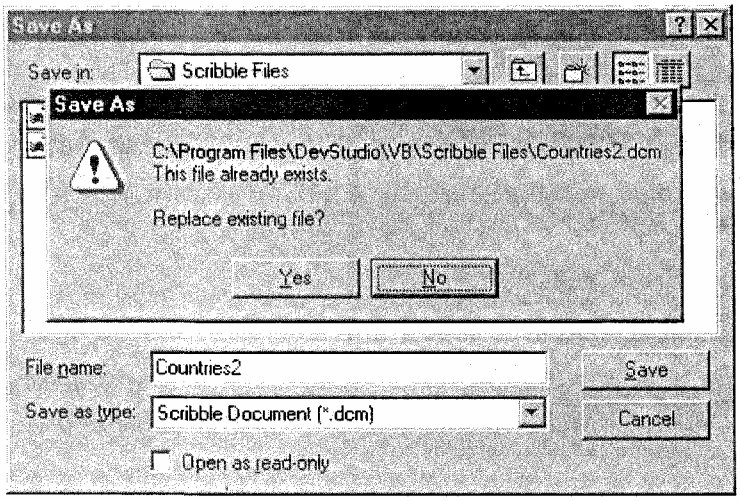


Figure 2.17 The screen at the end of the test the second time it is run.

However, when the tool comes to replay the exact input, after sending countries2, the next thing it sends is an attempt to select the *File* menu. Since this is not a valid input to this screen, Scribble will beep and redisplay the same screen. The automated test then tries to select the *Exit* option, causing Scribble to beep again and redisplay the same screen. The tool now thinks it is finished and exited from Scribble, so it may go on to start the next test case in the sequence. You can see how the automated testing will soon reach a state in which it will achieve nothing useful and might actually do some damage by deleting or changing data files that it should not.

The test tool is essentially 'blind' to what is really happening. All it can know about is what it has been told. Anything you want the automated test to know about, you have to tell it explicitly.

Again, the problem outlined here is but one of many possible ways in which a test can be tripped up. As with all such things, the problem is not insurmountable, but it takes time and effort to surmount it.

There are a number of alternative ways to overcome this problem. Here are three:

- Alter the script to respond to the extra prompt. The test case will now run the second and subsequent times, but will fail the first time, or if countries2.dcm has been deleted for any reason.
- Alter the script to check for the extra prompt and respond whenever it occurs. This enables us to have a script which will run regardless of whether countries2.dcm exists or not. Our script will contain an addi tional 'if statement along the lines of: 'if countries2.dcm exists, hit "Y".' Note that this will make our script more complex again.
- Alter the script to check for the file, and delete it if it exists before run ning the test case. Rather than 'hard-coding' the check for the existence of this file into the script at the point where the problem occurs, another possibility is to anticipate the problem at the beginning of the script. This may be better from a maintenance point of view, since something buried at the end of the script is more easily overlooked.

  In this case, we could put the check for the existence of the file at the beginning of the test script, and delete the file if it exists, so that the rest of the script will now run correctly. Now we don't have to check for the extra screen because we know it will not come up. Our script may now contain something like: 'if countries2.dcm exists, delete file coun-tries2.dcm.'

This example is typical of a number of things that can occur to trip up a test. It may be more efficient to collect all such similar things for a set of test cases into one place, where all the environment factors can be dealt with either for one test case or for a set of tests. This could be done as a separate script in the testing tool language, or it could be done as a completely separate exercise, possibly implemented in a command file or whatever is most familiar to the software engineers or testers. This is called pre-processing and is discussed in Chapter 6.

Which solution is best? This is another area where there are no answers, but choices to be made. The chosen solution will be a trade-off of many factors, including:

- implementation effort;
- future maintainability;
- test case failure analysis effort;
- debugging effort;
- this test case's ability to handle similar situations;
- robustness;
- what is done for other automated test cases.

2.5.2 Other verification requirements (often overlooked)

Having taken the environment considerations into account, our test is now finished, right? Well, what we now have is certainly a long way forward from our first attempt of simply capturing inputs.

The next question to ask is: is it a good test case?

Actually, this is the very first question we should have asked before we started automating it. As we discussed in Chapter 1, there is no point in automating poor test cases. However, the point we wish to make now concerns the comparisons we have implemented. Have we checked everything that we should? Will they reveal all the defects that could be found by this test case?

Often there are many facets to the outcome of a test case that we could check, which go beyond what we could illustrate with our simple Scribble example. Testers performing test cases manually will be making a multitude of checks, some consciously and probably many unconsciously. Automated tests will only check what they are programmed to check so it is important that we consider exactly what is programmed in and what can be left out.

Some of the outcomes we could consider for a lot of systems include:

- files that may have been created or changed (Were the correct files created? Were the correct files changed? Were they changed in the right way?);
- databases that may have been altered (Were the correct records altered? Were any alterations made to other records which should not have been touched?);
- other outputs, such as to printers, networks, or to other communication equipment (Were the right things sent to the right places? How can you be sure?);
- screen attributes may have been changed, possibly ones that are not visible such as whether a field is protected or not (Have the correct attributes been changed to the right new values?);

- internal data structures may have been created or altered, perhaps stored briefly before they are used by another program (Has the right information been put into that temporary storage? How can you check it?).

EXPERIENCE REPORT: BUG IN JET ENGINE DOWNS MICROSOFT'S HIGH-FLYER

Reports of a potentially serious bug, which corrupts database records in the Microsoft, Access application, have caused concern among its millions of users, writes Tom Forenski.

The bug is activated when editing database records and causes the edits to be posted to a record other than the one being viewed. The bug can be created simply by running one of Microsoft's own ivi/ardi. The hug is particularly devious in that everything appears to work correctly', reports Allen Browne. 'It is only when a record is checked later that the results become apparent. In addition, it only occurs when the form contains more than 262 records, so that the database works correctly initially, and the bug only surfaces as more data is added.'

The steps needed to reproduce the bug

are-delete a record on a form

without closing ihe form, search tor a different record using the bookmark properties of the form and recordset clone objects using the combo box wizard option 'Find a record on my form based on the value I selected in my combo box'

make data changes in the found record

The changes are then saved to a different database record than the intended one.

Microsoft has posted information on a workaround, and a fix would be released in the next service pack tot Access, said a Microsoft representative.

Source: Computrr Weekly, September 10, 1998

2.5.3 Ways of verifying file or database changes

How can we verify that these additional items have been correctly handled by the software? There are some choices for how we handle this.

One option is to add more verification to the script. We could check data values stored in a file or database by doing an additional read after we have modified something, to see whether the right information has been stored. This would be checked against the way we would expect to see that data displayed.

This does make the script more complex (again), and also more vulnerable to changes. If the layout of the display from the database were changed, all scripts that used this method would need to be updated, increasing the test maintenance burden.

Another possibility would be to use a comparison tool to search for differences between the correct version of the file and the actual file. This means that we would need to save a copy of the file as it should appear, in order to do the comparison. If this were a database, this may not be feasible.

As with many aspects of test automation, there are choices to be made, and the choice will affect many things. An effective test automation regime will have a good balance of checking things in a way that will be resistant if not immune to changes.

2.5.4 Where will all the files go?

There is one last thing to consider in implementing our example automated test case. In the process of creating the automated test case for Scribble, we have created several files. For example:

» countries.scp, our test script;

» countrics2.dcm, our expected output for post-execution comparison;

» countries2.dcm, our actual output;

» countries.dif, the list of differences produced by the comparator (if not in log);

« countries.log, the audit trail of what happened when the test ran.

All of these files must be given a home. Where should we put them? This is yet another decision to be made, and the results of that decision will affect the future maintainability of the tests. These issues are discussed in Chapter 5.

2,6 Conclusion: automated is not automatic

When you start implementing automated tests, you will find that you are running the (supposedly automated) tests manually. Automating some part of test execution docs not immediately give automatic testing; it is but the first step along a long road.

There arc many different approaches to automating testing, and generally speaking, the more elaborate the approach, the greater the end productivity (quicker to add new automated tests, less maintenance involved, etc.). Conversely, the simpler the approach, the less the end productivity. However, simple approaches have the advantage of comparatively low start-up costs and some benefits can be achieved fairly quickly, while elaborate approaches incur much larger start-up costs and the benefits are usually a much longer time coming.

It is important to choose an approach (or approaches) appropriate for the objectives of test automation. For example, if the tests are short term (perhaps because the software is to be used once and then thrown away) there is no point in using an approach that is designed to minimize long-term maintenance costs at the expense of upfront effort.

There is more to a test than just running it. Often there is some set-up and clear-up involved. This additional work also needs automating. Unfortunately, the easiest solution is often not the best solution. A standard approach is required in order to get the best balance between many alternative ways of solving the problems.

There is usually more verification to be done than first meets the eye. Automation can lead to better testing, if a comparator is used to compare things that might otherwise be overlooked if testing was done manually. However, this is by no means guaranteed.

The 'testware' must be designed to be maintainable and of high quality in order to achieve effective and efficient test automation, just as software must be designed to be maintainable and of high quality for effective and efficient use. This testware design process involves balancing many decisions concerning comparison, scripting, where files are stored, etc.

The commercial test execution tools are generic; one size fits all. However, each individual user is unique, and some parts of the generic solution may fit well, some not so well, and some may be downright damaging. In order to build an automated testing regime, you must tailor the generic tools to suit your own situation.

The key characteristics of a good automated regime are:

- it is very easy to select sets of tests to be run (the proverbial 'touch of a . button');
- those tests take care of all their own housekeeping such as environmen tal set-up and clear-down, i.e. the tests are self-sufficient;
- it is easier to add a new test to the automated pack than to run that test manually.

This can only be achieved through continuous improvement of the entire regime.

Summary

We have taken a very simple example test case for Scribble, our primitive word processor.

We looked at the different alternative starting points for automating this test, in terms of the current type of test documentation: ad hoc or unscripted, a vague manual test script, or a detailed manual test script.

Testing can be automated from any of these starting points, but is easiest from the detailed script. Recording is one way of getting the detailed test inputs into a form that a test execution tool can use, and can be used from any of the starting points. However, we do not recommend that you attempt to automate ad hoc testing, as it will not give you effective testing. Automating a chaotic process will just give you faster chaos.

We looked at the stages that you will go through in beginning to automate testing. The first step, especially if you have a capture replay tool, will

be to capture a test. What you actually capture is only the test inputs, however. Although this does give some advantages over typing the input manually, the test process still includes checking the screen manually as the test runs in order to see if the software is working correctly, as well as checking other things such as the file that was created.

Test comparison is normally also automated. This can be done by dynamic comparison within the script, or by post-execution comparison after the test has completed. With dynamic comparison, there are a number of choices to be made. The decisions made will affect how easy the tests are to maintain, how easy it will be to identify defects found by the test, and how likely the test is to miss defects that could have been found (unexpected changes).

Automating comparison still does not provide a good automated test, however. Other aspects must be taken into consideration. In our example, the file that was created by the test caused the test to fail the second time it was run automatically. This can be overcome, but this must also be built in to the test automation.

There may be other things that need to be verified in addition to what we can see on the screen, such as files or databases which have been altered by the test. We also need to decide where the files used and produced by the automated test will go. .

In a good test automation regime, tests are easy to run and self-sufficient, and it is easier to add a new automated test than to run that same test manually.

Scripting techniques

3.1 Introduction

Much of this chapter may seem familiar to anyone who has ever read a 'How to write maintainable programs' book. The techniques for producing maintainable scripts have many similarities to those for producing maintainable programs. Yet it seems that people who abandoned 'spaghetti code' many years ago end up producing 'spaghetti test scripts.' This chapter aims to insure that any test scripts that you produce, if they are not simply to be thrown away after a single use, are properly engineered.

3.1.1 Similarities with programming

Test scripts are a necessary part of test automation for interactive applications and, in some cases, non-interactive applications. As we demonstrated in Chapter 2, the information contained within a script can be extensive, and the more information there is the more there is to go wrong, update, and manage. Writing scripts is much like writing a computer program. Most test execution tools provide a scripting language that is in effect a programming language; thus, automating tests is a case of programming tests.

Of course programming tests correctly is no less difficult than programming software applications correctly and is usually about as successful. This means that no matter how clever we are at programming or testing there will be defects in the automated tests. The more programming, the more defects there will be. For the tests to work properly it is necessary to spend time testing the tests, debugging, and fixing them. This can become very time consuming, particularly because it is not limited to the first time the tests are automated. Changing the software usually forces some of the tests to be changed, for example, to cope with new options in the user interface, menu changes, user interactions, and changes to the expected results.

Test automation would be so much easier and quicker if it were not for the need to produce test scripts. This is a bit like saying that programming would be much easier if we did not have to write source code! However, this is exactly what we have been heading for during the past three or four decades. Third generation languages (3GLs) save us having to write a lot of assembler code, and fourth generation languages (4GLs) save us having to write as much 3GL code. We don't really want to have to write the code; rather we would prefer to only say what the system has to do and let it go ahead and do it (not having to tell it how to do the job). So it is with automated tests. We do not want to have to tell the tool down to every last keystroke how to run each test case, at least not all the time. Rather, we wish only to tell the tool what to do by describing the test case and not have to tell it how to execute it (a descriptive approach, not prescriptive). However, in the same way that 4GLs have not replaced 3GLs, and 3GLs have not entirely replaced assembler code, there are occasions when it is appropriate and necessary to write a script that specifies each individual keystroke. For example, when testing a real-time application, it can be useful to specify specific time delays between particular keystrokes.

The purpose of 3GLs and 4GLs is to increase productivity and make programming easier. There are different scripting techniques that seek to do the same (increase productivity, make test automation easier). Although test scripts cannot be done away with altogether, using different scripting techniques can reduce the size and number of scripts and their complexity. This chapter describes a number of different approaches to scripting, explaining the advantages and disadvantages of each technique and when it may be appropriate to use one in favor of the others.

3.1.2 Scripting issues in general

3.1.2.1 What test scripts are used for

Recording a test case being performed manually results in one (usually long) linear script that can be used to replay the actions performed by the manual tester. Doing the same for a number of test cases will result in one script for each test case. So if you have thousands of test cases you will end up with thousands of scripts. The script is really a form of computer program, a set of instructions for the test tool to act upon. Having one such program for every test case is not efficient since many test cases share common actions (such as 'add new client,' 'query client details,' etc.). This will lead to higher maintenance costs than the equivalent manual testing effort because every copy of a set of instructions to perform a particular action will need changing when some aspect of that action changes.

Most tools use a scripting language that offers far more power and flexibility than would ever be used by recording, but to benefit from it you have to edit the recorded scripts or write scripts (or rather, code scripts) from scratch. One of the benefits of editing and coding scripts is to reduce the amount of scripting necessary to automate a set of test cases. This is

achieved principally in two ways. One way is to code relatively small pieces of script that each perform a specific action or task that is common to several test cases. Each test case that needs to perform one of the common actions can then use the same script. The other way to reduce scripting is to insert control structures into the scripts to make the tool repeat sequences of instructions without having to code multiple copies of the instructions. Although these approaches inevitably cause us to code several more scripts initially, they are much smaller and therefore easier to maintain. Eventually, however, once a reasonably comprehensive set of scripts has been coded, new test cases can be added without the need to add more scripts so we enter a stage where thousands of test cases are implemented by hundreds of scripts.

Scripts may contain data and instructions for the test tool, including:

- synchronization (when to enter the next input);
- comparison information (what and how to compare, and what to com pare with);
- what screen data to capture and where to store it;
- when to read input data from another source, and where to read it from (file, database, device);
- control information (for example, to repeat a set of inputs or make a decision based on the value of an output).

The actual content of a script will depend on the test tool being used and the scripting techniques employed.

Like software, scripts are very flexible. There will usually be many ways of coding a script to perform a given task. The way a script is written usually depends on the skill of the person coding it but should also depend on its objective. A 'quick and dirty' approach may be to record as much as possible, or perhaps to copy parts of other scripts and cobble them together. A more thoughtful approach may involve some design work and coding from scratch.

Some scripting techniques involve elaborate constructs and logic while others are much simpler. The more elaborate the scripting technique used, the more upfront work there will be and the more effort will be needed on debugging the scripts. However, there will also be greater end productivity because there will be more reuse of scripts and they will involve less maintenance. The more tests you automate, the more worthwhile it is to put the effort into the more elaborate scripting techniques. This is illustrated in Figure 3.1.

However the scripts are created, what should concern us most is the cost of producing and maintaining them and the benefits we can obtain from using them. If a script is going to be reused by a lot of different tests that are expected to have a long life, then it will be worth ensuring that the script works well and is easy to maintain. If, however, the script is to be used for only one test case that will be discarded as soon as the testing has been completed, then there is little point in spending much effort creating it.

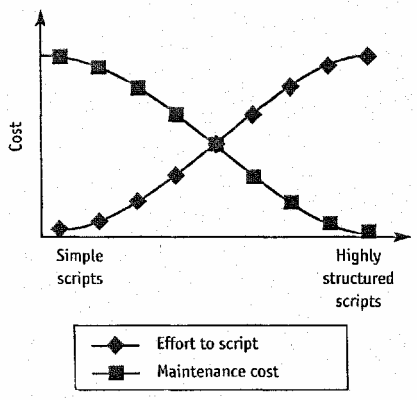


Figure 3.1 Script development effort versus maintenance effort.

It is worth considering which scripting techniques you arc likely to use when you are choosing a tool so you can check for appropriate support (see Chapter 10).

3.1.2.2 Good versus bad scripts

Since scripts form such a vital part of most test automation regimes, it seems sensible that we should strive to insure that they are good. Certainly we can be sure that nobody would wish to be associated with bad scripts. So what is a good script?

The easy answer to this question is 'a script that is fit for purpose' but that doesn't amount to a very comprehensive answer. Does this mean that a good script is one that does what it is supposed to do reliably and is easy to use and maintain? Often yes, but there are circumstances when scripts do not have to be particularly easy to use or maintain (such as the one-off automated test cases described earlier) though it is difficult to think of any circumstances when scripts need not do what they are supposed to or be unreliable. If you arc not concerned with ease of use or maintainability for any of your scripts then much of what we have to say in this section will be of little interest to you.

Given that good scripts should be easy to use and maintain we can describe a few characteristics or attributes that will have a significant bearing on how easy to use or maintain a script really is. Rather than consider individual scripts, we will compare two sets of scripts that perform the same set of test cases in Table 3.1.

When test automation begins, you will start out with a small number of scripts and it will not be difficult to keep track of them, but this task will become more difficult as the number of scripts increases. Ways of organizing scripts are described in Chapter 5.

Table 3.1 Comparison of good and poor sets of test scripts that perform the same set of test cases.

Attribute	*Good set of test scripts*	*Poor set of test scripts*
Number of scripts	Fewer (Less than one script for each test case)	More (at least one script for each test case)
Size of scripts	Small - with annotation, no more than two pages	Large - many pages
Function	Each script has a clear, single purpose	Individual scripts perform a number of functions, typically the whole test case
Documentation	Specific documentation for users and maintainers, clear, succinct and up-to-date	No documentation or out-of-date; general points, no detail, not informative
Reuse	Many scripts reused by different test cases	No reuse; each script implements a single test case
Structured	Easy to see and understand the structure and therefore to make changes; following good programming practices, well-organized control constructs	An amorphous mass, difficult to make changes with confidence; spaghetti code
Maintenance	Easy to maintain; changes to software only require minor changes to a few scripts	Minor software changes still need major changes to scripts; scripts difficult to change correctly

3.1.2.3 Principles for good scripts

What is it that makes a script 'easy to use' or 'easy to maintain'? There is bound to be a subjective element to the answers to these questions, as different people will have varying preferences as to the style, layout, and content of scripts. However, in our view the following principles provide a guide.

Scripts should be:

- annotated, to guide both the user and the maintainer;
- functional, performing a single task, encouraging reuse;
- structured, for ease of reading, understanding, and maintenance;
- understandable, for ease of maintenance;
- documented, to aid reuse and maintenance.

These basic principles apply whatever scripting technique is used, but they may be implemented in different ways. For example, all scripts should be annotated, but the annotation may differ in style and format from one regime to another.

3.1.2.4 'Easy to read' scripts?

Occasionally a tool vendor will claim that its tool generates recorded scripts that arc 'easy to read' or even 'self documenting.' Such claims are misleading and very short-sighted. Let us deal with these claims once and for all.

A 'raw' recorded script consists of a whole sequence of inputs for the test and may also include some verification. Some tools under user instruction can insert the verification instructions as the recording takes place. The end result is usually a long script, perhaps hundreds of lines covering several pages, that does not have a useful structure. It could be likened to a book that has no chapters or section headings but just page after page of text. Finding a particular topic in such a book would be tiresome to say the least, as you would have to read through it until you found the required text. So it is with a recorded script. The reason we would wish to look at a recorded script anyway is usually because we need to make a change to it, perhaps to insert some verification or control instructions, or to change a particular sequence of actions to reflect changes in the software under test. In order to make these changes correctly it will be necessary for us to find the relevant part of it, and the longer the script is, the more difficult this will become. It would also be wise for us to gain an understanding of the script as a whole, to help insure that any changes we make will not have adverse affects on other parts of the script or indeed any other scripts that use this one.

Of course, finding things in a script and gaining an understanding of it will be easier for us if we are familiar with the scripting language. It will also be easier if there is some useful documentation in the script telling us what it does and how it does it. Script annotation, comments embedded throughout the script that say what is happening at each stage, will be the most helpful. Some tools automatically embed comments in the scripts they generate while recording and this is sometimes what the tool vendors allude to when they claim their tools' scripts are 'easy to read.'

However, 'easy to read' can be taken to mean different things. Scripts usually comprise keywords that are the same as they appear in the English language, such as 'Type,' 'Button,' and 'Click.' Fair enough, but is this 'ease to read' what we want? Certainly we do not want 'difficult to read' scripts (perhaps using acronyms and symbols rather than proper words, although this would depend on the person doing the reading, since an experienced author may prefer familiar abbreviations and acronyms for ease and speed of typing and reading).

When we wish to make changes to the script, its readability becomes very important, but readability in this sense means easy to understand. This leads us to the second common claim.

3.1.2.5 'Self-documenting' scripts?

'Self documenting' is a rather mythical term. To document a script means to give written information supporting or detailing it. We interpret this as written information that is useful and in addition to what is already there. The information that can usefully be written down somewhere includes the following:

- the raw content of the script (inputs and sometimes expected outputs);
- the purpose of the script (what this script does, the actions it performs);
- user information (what information has to be passed to the script, what information it returns, what state the software under test has to be in when the script is called, and what state the software will be left in when the script ends);
- implementation information (additional information helpful to maintainers, such as an explanation of why it was implemented in a particular way or references to similar scripts that may also need changing);
- annotation (comments embedded throughout the script to say what is happening at each logical step in terms of the actions being performed on or by the software under test).

It is only the first item above that a tool can generate. The only annotation that can be generated 'automatically' is limited to 'click the OK button,' 'focus on window "Payments"' and 'input the string "John Doe" in the field "name".' Such information is often rather obvious anyway, so the annotation is generally not adding any value. This type of annotation certainly is not sufficient if the script is to be used and maintained effectively and efficiently.

None of the other information can be generated automatically. But if you want understandable scripts, someone must produce this information, generally a job for the script's author. To add this information to the script, this person needs to be able to edit the script itself so he or she needs to have some technical skills (unless the tool supports easy insertion of comments during script creation). If the people recording the scripts are not technical, then adequate support must be provided to them.

Simply recording a long test is not by any means the end of the story; there is a lot more work to be done. However, a recording is often the place where test automation starts and is sufficient in a few instances.

3.1.3 Test case design and implementation

3.1.3.1 Test case design

There is not always a direct relationship between an automated script and a test case. In fact, what is taken to be a test case can vary enormously. For some people a single test case may last only a few seconds or minutes but for others a test case can last for several hours. Long test cases can usually be considered as a series of individual tests, as shown in Figure 3.2. The rectangles in this figure each represent a single test case and include 'in' and 'out' activities such as invoking the software under test and terminating its

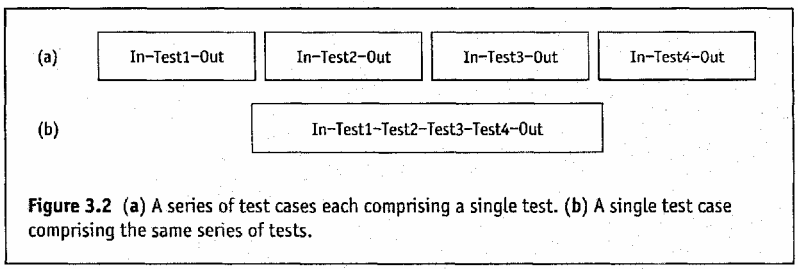


Figure 3.2 (a) A series of test cases each comprising a single test. (b) A single test case comprising the same series of tests.

execution. In Figure 3.2(a) each test is implemented as a separate test case, whereas in Figure 3.2(b) a single test case executes all four tests. It is important to realize that tests 2, 3, and 4 are performed under different conditions, so, strictly speaking, they are different tests. However, the inputs and expected outcomes would be the same for the tests, whichever way they were performed.

We use the term 'test case' to mean a scries of one or more tests that can be meaningfully distinguished from other such series. For example, running one test case on our Scribble program may involve starting the program, opening a particular document, performing some edits, saving the document, and exiting the program. However, some people's perception of a test case would not include starting and exiting the program. Thus, if a set of test cases were performed, the program could be started, all the test cases run and then the program terminated.

It may be helpful to think of a single automated test case as being the shortest sequence of tests that collectively are assigned a single status. Commercial test tools typically assign a 'passed' or 'failed' status to some amount of testing but they are not so consistent when it comes to naming this unit of testing. For example, test case, test procedure, and test script are used by different tools to mean the same thing. We will use the term test case.

3.1.3.2 Jest case implementation

Regardless of how long or short, complex or simple test cases are, there is not necessarily a direct relationship between the test cases and the scripts used by the test tool in performing them. Although a few tools force their users into defining separate 'scripts' for each test case, this is not generally helpful when implementing a test automation regime of any substance.

At first it may seem sensible to implement each test case using a single script. Indeed, this is actually encouraged by some tools. However, as we shall show, this is not going to lead to an effective and efficient test automation regime. Implementing automated tests in this way will always be costly, and the cost of maintaining them will continue to grow and probably become greater than the cost of performing the same tests manually.

The following section describes a number of scripting techniques that take us into script programming. It is only by breaking the 'one test case one test script' relationship that we will ever achieve a good automated testing regime.

3.1.4 Recommendations for script documentation

We recommend that the documentation for each script is placed in the header of the script itself and that it should be as concise as possible. These two points will help whenever the script is changed, since whoever is making the change will be more likely to update the documentation as well as the body of the script. The layout of the script documentation is a matter of style that may already be governed by existing coding standards or conventions (if these work well then use them; if not then change them). In our view it is important that everyone who writes or edits scripts conforms to the same conventions. Everyone will know what information to include in the documentation, where to put it, and how to present it. They will also know where to look for particular information. Another advantage of everyone conforming to the same standards or conventions is that it makes it easy to copy the information from each script into a separate document such as a catalog of all shared scripts. Such a task can be automated easily using one or two suitable text manipulation tools, and indeed it is best automated since it can be run 'at the touch of a button' to produce an up-to-the-minute catalog.

We suggest a script header layout like that shown in Figure 3.3. This can be used as a template for all scripts or it can be adapted for different types of scripts (though we do prefer to see one common header for all scripts). The actual layout is not as important as the content but we do recommend all our scripts are laid out in a consistent way. Note that the semicolon at the start of each line in Figure 3.3 is used here as a comment marker to tell the test tool to ignore the whole line. How comments are identified in your scripts will depend on the tool you use.

```
;                Test Automation Script
; SCRIPT: ; AUTHOR: ; DATE :

; PURPOSE :

; PARAMETERS : ;  PRE- AND POSTCONDITIONS: ;

MAINTENANCE
; AMENDMENTS : ;  <id> <name> <date> <what & why>
```

Figure 3.3 Our suggestion for a script header. This contains all the information necessary to understand what a script does and how to use it. It also contains any information that would benefit someone who has to maintain the script.

Everything in this template has a purpose but they will not all be relevant to everyone. The template contains a series of keywords which serve to remind the script writer what information to document (and where to put it) and make it easy to automate the task of copying the information into a separate file. The title Test Automation Script' simply tells us what this is. It may seem like overkill but it can be useful in environments where there are a lot of different types of file and all with numerous uses. It would be improved if it included the name of the test tool (particularly if the scripting language is unique to the tool). Similarly, the keyword 'SCRIPT' should be followed by the name of the script. This will be redundant if printouts automatically have the filename included. The purpose is to avoid the problem of finding a printout without the filename on it - it can be time consuming, if not difficult, to find out the script's name.

The 'AUTHOR' and 'DATE' keywords should be followed by the name of the scriptwriter and the date when it is written. Having the author's name clearly makes it possible to identify who wrote the script, something that can be useful to know (to ask questions or to ask for help, for example). The date may be redundant, particularly if there is a good configuration management system in place that can provide this information.

A brief statement as to the purpose of the script should follow the 'PURPOSE' keyword. Usually this will not need to be more than a single, short sentence. If it is any more then it is likely that the script is performing too much and may be better divided into two or more smaller (and possibly more reusable) scripts. We prefer the keyword 'purpose' rather than the ubiquitous 'description' since it discourages totally worthless statements like 'This is a script.'

The 'PARAMETERS' keyword heads up a section of the header that describes any parameters that arc accepted or returned by this script. It may be helpful to distinguish between parameters that are passed in and those that are passed out or returned. The 'PRE- AND POSTCONDITIONS' section informs users of the script what state the software under test has to be in at the time the script is called and what state the software will be left in when the script has completed. This is particularly important information if users of the script are going to be able to use it correctly first time, every time. For example, the script may expect the software application to have a specific dialog box displayed but will leave a different one displayed when it exits. If the person using this script did not know this, the script may be called when the required dialog box was not displayed. This could cause the test case to fail (but not necessarily at the point the shared script was called, thereby making it more difficult to identify and correct the problem).

The 'MAINTENANCE' section contains any information that would be useful to people who will be maintaining the script. Most of the time this will be blank (rather than removed altogether) since the user information given in the rest of the header together with the script annotation will be sufficient. However, sometimes scripts have to be implemented in a strange or unusual way in order to avoid some problem or other (such as a tool feature that does not work correctly or an idiosyncrasy of the software application).

The 'AMENDMENTS' section contains details of all changes made to the script since it was first completed. The 'id' can be used to mark all the lines in the script that are altered or added in order to make the amendment. Again, this section may be redundant if there is a good configuration management system in place that can provide this information.

3.2 Scripting techniques

We now look at some different scripting techniques and their uses. These techniques are not mutually exclusive. In fact, the opposite is true - they can, and most likely will, be used together. Each technique has its own advantages and disadvantages that affect the time and effort it takes to implement and maintain the test cases that the scripts support.

However, it is not which techniques are used that is most significant; rather, it is the overall approach taken by the whole regime in implementing test cases supported by the scripts. This has the greatest impact. There arc two fundamental approaches to test case implementation, prescriptive and descriptive. These are discussed later in this chapter, after we have seen what different scripting techniques look like.

In the rest of this section, we use simple example scripts to illustrate the points we wish to make, using a pseudo-scripting language which should be understandable to all readers, whether or not you have actually used a tool. In reality, the actual tool scripts would be more complex than the examples we show. For simplicity, we show only the minimum script features that allow us to explain the different scripting techniques.

The scripting techniques described are:

- linear scripts;
- structured scripting;
- shared scripts;
- data-driven scripts;
- keyword-driven scripts.

3.2.1 Linear scripts

A linear script is what you end up with when you record a test case performed manually. It contains all the keystrokes, including function keys, arrow keys, and the like, that control the software under test, and the alphanumeric keys that make up the input data. If you use only the linear scripting technique - that is, if you record the whole of each test case -each test case will be replayed in its entirety by a single script.

A linear script may also include comparisons, such as 'check that the error message *Position not valid* is displayed.' Adding the comparison instructions may be done while the test case is being recorded (if the test tool allows this) or it may be done in a separate step afterwards, perhaps while replaying the recorded inputs from the script.

Note that a test case that takes 10 minutes to run manually may take anywhere from 20 minutes to 2 hours to automate with comparison. The reason for this is that once it is recorded, it should be tested to make sure it will play back - it is surprising how often this is not the case! Then it may need to be played back again while the specific comparison instructions are added, and the new script with comparison embedded should also be tested. If it is not correct, the test script then needs to be debugged and tested again. The more complex the application and test case, the longer this process is likely to take.

3.2.1.1 Advantages of linear scripts

Linear scripts have advantages that make them ideal for some tasks. The advantages are:

» no upfront work or planning is required; you can just sit down and record any manual task; » you

can quickly start automating;

- it provides an audit trail of what was actually done;

» the user doesn't need to be a programmer (providing no changes are required to the recorded script, the script itself need not be seen by the user);

- good for demonstrations (of the software or of the tool).

3.2.1.2 When to use linear scripts

Almost any repetitive action can be automated using a linear script. There are some situations where a simple linear script is the best scripting technique to use. If a test case will only be used once, for example, to investigate whether a test execution tool works in a given environment, then there is no point in investing any more effort than necessary in such a script since it will be thrown away.

Linear scripts can be used for demonstrations or training. When you want to show the features of the software to a potential customer, and don't want to have to be concerned with typing in exactly the right keystrokes when you are feeling a little nervous anyway, replaying a recorded script will reproduce the keystrokes exactly.

Linear scripts can be used to automate edits to update automated tests. Any given update will probably only be done once, so a throwaway script is all that is needed. Linear scripts can be used to automate set-up and clear-up for tests, or to populate a file or database by replaying an input sequence.

Linear scripts can be useful for conversions. If some part of the system has been changed but without changing the working of the system from the user's perspective, recording live data one day, replacing the software or hardware, and then replaying the day's traffic can give an initial level of confidence that the new system generally works. This approach has been used successfully for benchmark testing for Year 2000 conversion.

3.2.1.3 Disadvantages of linear scripts

Linear scripts do have a number of disadvantages, particularly with respect to the building of a long-term test automation regime:

- the process is labor-intensive: typically it can take 2 to 10 times longer to produce a working automated test (including comparisons) than run ning the test manually;
- everything tends to be done 'from scratch' each time;
- the test inputs and comparisons are 'hard-wired' into the script;
- there is no sharing or reuse of scripts;
- linear scripts are vulnerable to software changes;
- linear scripts are expensive to change (they have a high maintenance cost);
- if anything happens when the script is being replayed that did not happen when it was recorded, such as an unexpected error message from the network, the script can easily become out of step with the software under test, causing the whole test to fail.

These disadvantages make the sole use of linear scripts to automate testing an impractical approach for long-term and large numbers of tests. Every test to be automated will take as much effort as the first, and most scripts will need some maintenance effort every time the software under test changes.

3.2.2 Structured scripting

Structured **scripting** is parallel to structured programming, in that certain special instructions are used to control the execution of the script. These special instructions can either be control structures or a calling structure.

There are three basic control structures supported by probably all test tool scripting languages. The first of these is called 'sequence' and is exactly equivalent to the linear scripting approach we described earlier. The first instruction is performed first, then the second, and so on. The other two control structures are 'selection' and 'iteration.'

The selection control structure gives a script the ability to make a decision. The most common form is an 'if statement that evaluates a condition to see if it is true or false. For example, a script may need to check that a particular message has been displayed on the screen. If it has it can continue; if it has not it has to abort. In this example the condition is whether or not a particular message has been displayed.

The iteration control structure gives a script the ability to repeat a sequence of one or more instructions as many times as required. Sometimes referred to as a 'loop,' the sequence of instructions can be repeated a specific number of times or until some condition has been met. For example, if a script were required to read records containing data from a file, the sequence of instructions would perform the read and process the information in some

way. This sequence can then be repeated until all the records in the file have been read and processed.

In addition to the control structures, one script can call another script, i.e. transfer control from a point in one script to the start of another sub-script, which executes and then returns to the point in the first script immediately after where the sub-script was called. This mechanism can be used to divide large scripts into smaller, and hopefully more manageable, scripts.

Introducing other instructions for changing the control structure gives even more opportunities for not only increasing the amount of reuse we can make of scripts but also increasing the power and flexibility of the scripts. Making good use of different control structures leads to maintainable and adaptable scripts that will in turn support an effective and efficient automated testing regime.

Making good use of different structured scripting techniques does require programming skills. It is not necessary to have years of programming experience or indeed knowledge of numerous programming languages. We would suggest that the minimum requirement is someone who is interested in learning and is given both the time and resources (such as training) to learn. If experienced programmers can be brought into the test automation project, so much the better.

Figure 3.4 shows a script for part of our Scribble test. After entering the filename and clicking the OK button in the *Save As* dialog box, the script then checks to see if there has been a *Replace existing file?* message output. If there has, it will click the *OK* button, but if not it will continue in the normal way.

This example has been simplified. In practice a script may have to wait a few seconds to give Scribble time to display the message. The script probably would also need to move the focus to the message box before it

Part of the Scribble test script

```
SelectOption 'File/Close' FocusOn ' Close '
LeftMouseClick 'Yes' FocusOn 'Save As' Type
countries2 LeftMouseClick 'Save'
If Message = 'Replace existing file?'
LeftMouseClick 'Yes' End if
FocusOn 'Scribble' SelectOption
'File/Exit'
```

Figure 3.4 This script fragment contains a check for a *Replace existing file?* message using an 'If instruction (shown in bold). If this message is displayed the OK button will be clicked. If the message is not displayed the script will continue as usual.

could read the message. These considerations further complicate the script but this may be well worth doing since it makes the script more flexible and robust.

The main advantage of structured scripting is that the script can be made more robust and can check for specific things which would otherwise cause the test to fail, even though the software was working correctly as shown in Figure 3.4. A structured script can also deal with a number of similar things that need to be repeated in the script, using a loop. A structured script can also be made modular by calling other scripts.

However the script has now become a more complex program and the test data is still 'hard-wired' into the script.

3.2.3 Shared scripts

Shared **scripts,** as the name implies, are scripts that are used (or shared) by more than one test case. Of course, this implies that we require a scripting language that allows one script to be 'called' by another but this is more or less standard across all test execution automation tools.

The idea of this is to produce one script that performs some task that has to be repeated for different tests and then whenever that task has to be done we simply call this script at the appropriate point in each test case. This gives two distinct advantages: first, we do not have to spend time scripting (writing or recording the actions required); and second, we will have only one script to change in the event that something concerning the repeated task changes.

One of the features of current development tools is the ease with which graphical development environments can change the user interface to a system. However, the aspects that make this so attractive to users and developers are also the aspects that can be most destabilizing to automated testing. The use of shared scripts is one step towards building automated tests that can keep up with rapidly changing software, i.e. that will not require extensive maintenance.

For example, rather than have the navigation repeated in a number of scripts, when the commands to navigate to a particular place in the application are identical, there can be a single shared script that does the navigation for all tests. The shared script contains the inputs to the application that cause it to navigate to a particular screen which is the starting point for a number of tests. Each of the tests calls the navigation script and then runs the detailed test. When the tests are completed, a common script would be called to navigate back out to the main menu.

3.2.3.1 Going from a linear to a shared script: example

An example of shared scripts for our example Scribble test case is shown in Figure 3.5. A shortened linear script Scribblel shown on the left has been divided into the three separate scripts shown on the right-hand side. The first of these new scripts, ScribbleOpen, invokes the Scribble application and causes it to open a document, the name of which is passed into the script at the time it is called. Similarly, the last of these new scripts, ScribbleSaveAs, causes

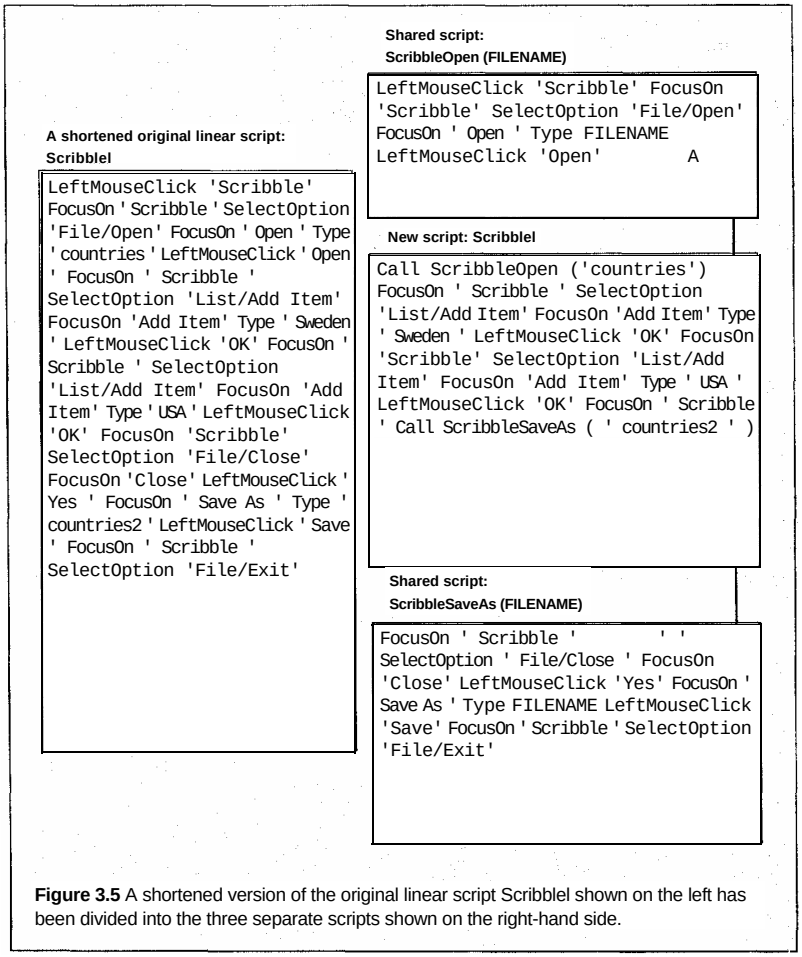


Figure 3.5 A shortened version of the original linear script Scribble1 shown on the left has been divided into the three separate scripts shown on the right-hand side.

Scribble to save the current document using a name that is passed into it at the time it is called. The third script (the new Scribble1) first calls ScribbleOpen then performs the testing actions before calling ScribbleSaveAs to finish.

Of course it will take longer to implement the shared scripts version of the test case since we have to create three scripts instead of just the one. If we create them by editing a recorded script, the two shared scripts will need to be changed so the filename they enter is not the one used at the time they were recorded but can be specified at the time the script is called. This is usually achieved by replacing the literal filename with a variable name (here we have used the variable name FILENAME), which can be thought of as a place marker that will be substituted for the real filename by the test tool when the script is called. The new test script Scribble1 will also have to be edited to insert the calls to the two shared scripts.

Scribble2 (linear)

```
Lef tMouseClick ' Scribble ' FocusOn
' Scribble ' SelectOption
'File/Open' FocusOn 'Open' Type
' countries ' Lef tMouseClick
'Open' FocusOn ' Scribble '
SelectOption 'List/Add Item'
FocusOn 'Add Item' Type ' France
' Lef tMouseClick 'OK' FocusOn '
Scribble ' SelectOption 'List/Add
Item' FocusOn 'Add Item' Type '
Germany ' Lef tMouseClick 'OK'
FocusOn ' Scribble ' SelectOption
'File/Close' FocusOn 'Close' Lef
tMouseClick ' Yes ' FocusOn ' Save As
'Type 'test2' LeftMouseClick 'Save
' FocusOn 'Scribble'
SelectOption 'File/Exit'
```

ScribbleZ (using shared scripts)

```
Call ScribbleOpen( countries )
FocusOn 'Scribble' SelectOption
'List/Add Item' FocusOn 'Add Item'
Type 'France' Lef tMouseClick 'OK'
FocusOn 'Scribble' SelectOption
'List/Add Item' FocusOn 'Add Item'
Type ' Germany ' Lef tMouseClick 'OK'
FocusOn 'Scribble' Call
ScribbleSaveAs ('test2')
```

Figure 3.6 The second and subsequent test cases can be implemented much more easily since the shared scripts can be reused.

Actually, calling the two shared scripts 'shared' is a bit misleading at this time since they are not shared at all! Only one script calls them. However, as soon as we start to implement another similar test case we can use them a second time. Figure 3.6 shows a new control script, Scribble2, that implements a slightly different test. This is much shorter than the linear script for this test case so it should not take as long to implement. We can carry on implementing more test cases that each call these two shared scripts and others beside. If the user interface of Scribble changes such that one of the shared scripts needs changing we will have only the one script to change. Had we used different linear scripts to implement the whole of each test case then we would have to change every one.

This approach is useful for any actions that are done often or that are susceptible to change. Navigation is one example, but any scripted activity that is repeated often may well be better in a common script, particularly if it is a complex or lengthy sequence.

3.2.3.2 Types of shared scripts

There are two broad categories of shared script, those that can be shared between tests of different software applications or systems, and those that can only be shared between tests of one software application or system. Some examples of these two categories are given in Table 3.2. Note that application-independent scripts may be more useful long term, and are worth putting additional effort into.

Table 3.2 Types of shared scripts,

Application-specific scripts	*Application-independent scripts*
Menus	Login and logout
Individual screen/window routines	Synchronization
Non-standard controls	Logging
Navigation	Input retrieval
	Results storage
	Error recovery
	Data-driven shell
	Checking or comparison

3.2.3.3 Advantages and disadvantages of shared scripts

The advantages of shared scripts are:

» similar tests will take less effort to implement;

» maintenance costs are lower than for linear scripts;

» eliminates obvious repetitions;

» can afford to put more intelligence into the shared scripts. For example, on login if the network is busy, wait *2* minutes and try again. It wouldn't be worth doing that in hundreds of scripts, but is well worth doing in one.

The shared scripting approach is good for small systems (for example, simple PC applications and utilities) or where only a small part of a large and stable application is to be tested using relatively few tests (a few tens of tests, certainly no more than a few hundred).

There are also some disadvantages to shared scripts:

« there are more scripts to keep track of, document, name, and store, and if not well managed it ma^ be hard to find an appropriate script;

« test-specific scripts are si ill required for every test so the maintenance costs will still be high;

» shared scripts are often specific to one part of the software under test.

3.2.3.4 How to get the best out of shared scripts

In order to reap the benefits of shared scripts, you have to be disciplined. You have to insure that all tests actually do use the shared scripts where they arc appropriate. We were once told that programmers will look for a reusable function for up to two minutes before deciding to implement their own version. We are sure this is much the same for automated test script writers. Knowing that there is, or at least should be, a reusable script to do a particular job is one thing; being able to find it is another. If it cannot be found quickly people will write their own version of it and then we will have a much more difficult task with maintenance updates. When the script

needs changing the person undertaking the maintenance will believe that it is only the shared script that needs changing. The fact that there is another script that needs editing may not be realized until the tests are run. Even then it may not be obvious what the problem is.

Some form of reusable script library will help matters considerably. This has to be set up and managed, and although it will not be a full-time job, it is likely to take a significant amount of effort in the early days. Script libraries are an aspect of testware architecture, which is covered by Chapter 5.

Another aspect of shared scripts that needs careful attention is their documentation. The scripts need to be documented in such a way that makes it easy for testers to determine what each one does (to tell whether this is really what they need) and how to use it. Software documentation in general is notoriously poor. Where it exists it is often out of date or just plain wrong. If these traits are carried into the documentation of our scripts then automation will suffer from unnecessarily high implementation and maintenance costs.

A good way to try to avoid these problems is to adopt conventions and define standards for scripts. These will form a part of the overall test automation regime. They will not only help insure documentation is considered but also will guide new script writers as to how to go about creating scripts (and how to make new shared scripts available for others to use).

3.2.4 Data-driven scripts

A data-driven scripting technique stores test inputs in a separate (data) file rather than in the script itself. This leaves the control information (e.g. menu navigation) in the script. When the test is executed the test input is read from the file rather than being taken directly from the script. A significant advantage of this approach is that the same script can be used to run different tests.

For example, one test for an insurance system will enter the details of a new insurance policy and verify that the database has been updated correctly. A second test would do the same thing except it would use a different insurance policy. It therefore requires the same instructions but different input and different expected outcomes (the values that describe a different insurance policy). These two tests can be implemented using a single test script and a data file.

3.2.4.1 Data-driven example 1: using the same script with different data Figure 3.7 shows how this could work with our Scribble test case at a basic level. (Figure 3.8 shows a more sophisticated approach.) Rather than having one script for each test case we can now implement numerous test cases with a single script. We will refer to this single script as the control script because it controls the execution of a number of test cases. Note that each of the test cases we are showing here only adds two names to the list of names in the file 'countries.' This is because we have shortened the original