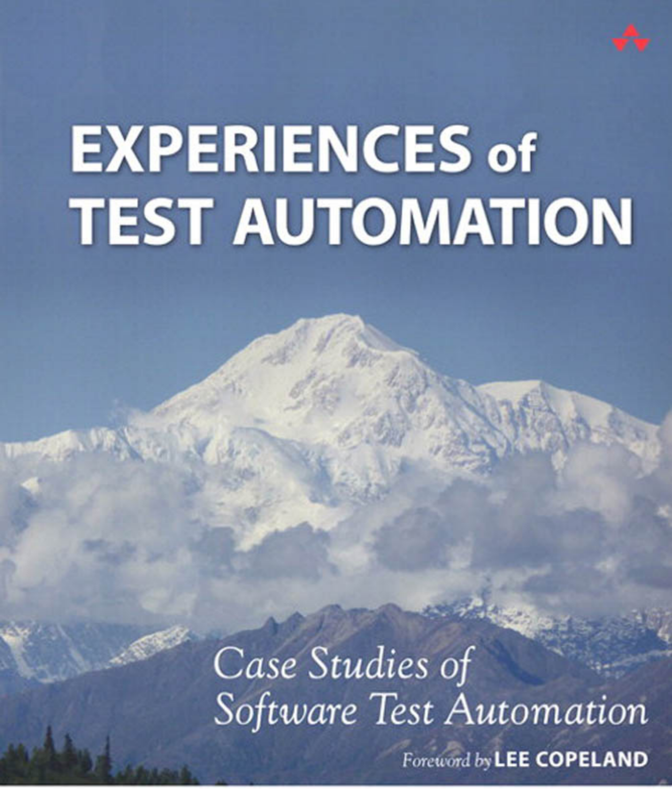

EXPERIENCES of TEST AUTOMATION

Case Studies of Software Test Automation

Foreword by **LEE COPELAND**

DOROTHY GRAHAM · MARK FEWSTER

Praise for *Experiences of Test Automation*

"What you hold in your hands is a treasure trove of hard-won knowledge about what works and what doesn't in test automation. It can save you untold hours and costs by steering you away from paths that lead nowhere and guiding you towards those that lead to success."

—Linda Hayes

"From tools to methodology, Dorothy Graham and Mark Fewster weave a compelling set of stories that provide a learning experience in automation. This comprehensive tome is the first of its kind to take the reader deep into the world of automated testing, as depicted by case studies that show the realities of what happened across a multitude of projects spanning a wide variety of industries and technology environments. By identifying similarities and repeated themes, the authors help the reader focus on the essential learning lessons and pitfalls to avoid. Read this book cover to cover for inspiration and a realization of what it takes to ultimately succeed in test automation."

—Andrew L. Pollner, President & CEO of ALP International Corporation

"Many years after their best-seller *Software Test Automation*, Mark Fewster and Dorothy Graham have done it again. Agile methodologies have given test automation a dominant presence in today's testing practices. This is an excellent, highly practical book with many well-documented case studies from a wide range of perspectives. Highly recommended to all those involved, or thinking about getting involved, in test automation."

—Erik van Veenendaal, Founder of Improve Quality Services and vice-chair of TMMi Foundation

"This book is like having a testing conference in your hand, with a wealth of case studies and insights. Except that this book is much cheaper than a conference, and you don't have to travel for it. What impressed me in particular was that it is all tied together in a concise 'chapter zero' that efficiently addresses the various aspects I can think of for automation success. And that is something you will not get in a conference."

—Hans Buwalda

"An exciting, well-written, and wide-ranging collection of case studies with valuable real-world experiences, tips, lessons learned, and points to remember from real automation projects. This is a very useful book for anyone who needs the evidence to show managers and colleagues what works—and what does not work—on the automation journey."

—Isabel Evans, FBCS CITP, Quality Manager, Dolphin Computer Access

"*Experiences of Test Automation* first describes the essence of effective automated testing. It proceeds to provide many lifetimes worth of experience in this field, from a wide variety of situations. It will help you use automated testing for the right reasons, in a way that suits your organization and project, while avoiding the various pitfalls. It is of great value to anyone involved in testing—management, testers, and automators alike."

—Martin Gijsen, Independent Test Automation Architect

"This offering by Fewster and Graham is a highly significant bridge between test automation theory and reality. Test automation framework design and implementation is an inexact science begging for a reusable set of standards that can only be derived from a growing body of precedence; this book helps to establish such precedence. Much like predecessor court cases are cited to support subsequent legal decisions in a judicial system, the diverse case studies in this book may be used for making contemporary decisions regarding engagement in, support of, and educating others on software test automation framework design and implementation."

—Dion Johnson, Software Test Consultant and Principle Adviser to the Automated Testing Institute (ATI)

"Even with my long-established 'test automation won't work' stance, this book did make me pause and ponder. It opened my mind and gave me a few 'oh, I hadn't thought of that' moments. I would recommend this book as an initial reference for any organization wanting to introduce test automation."

—Audrey Leng

"This book is a stunning achievement. I believe that it is one of the best books ever written in test automation. Dot and Mark's approach presenting 28 case studies is a totally new concept including eye-catching tips, good points, and lessons learned. The case studies are coming from life experiences, successes and failures, including several aspects of automation, different environments, and a mixture of solutions. Books are 'the' source of wisdom, and what a good idea for using storytelling to increase our learning through triggering our memories. This book is a must for everyone who is thinking of or involved in test automation at all levels. It is truly unique in its kind."

—Mieke Gevers

Experiences of Test Automation

This page intentionally left blank

Experiences of Test Automation

Case Studies of Software Test Automation

Dorothy Graham
Mark Fewster

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Graham, Dorothy, 1944-
Experiences of test automation : case studies of software test automation / Dorothy Graham, Mark Fewster.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-75406-6 (pbk. : alk. paper)
1. Computer software—Testing—Automation—Case studies. I. Fewster, Mark, 1959- II. Title.
QA76.76.T48G73 2011
005.3028'7—dc23
2011040994

ISBN-13: 978-0-321-75406-6
ISBN-10: 0-321-75406-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Second printing, August 2012

To my husband, Roger, for your love and support,
your good ideas, and for making the tea!
And to Sarah and James, our wonderful children.
—Dot Graham

To my wife, Barbara, for the good times we've shared.
And to my terrific son, Rhys, for the good times you bring.
—Mark Fewster

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Automated testing—it's the Holy Grail, the Fountain of Youth, and the Philosopher's Stone all rolled into one. For decades, testers have looked to automated testing for relief from the drudgery of manual testing—constructing test cases and test data, setting system preconditions, executing tests, comparing actual with expected results, and reporting possible defects. Automated testing promises to simplify all these operations and more.

Unfortunately, successful, effective, and cost-effective automated testing is difficult to achieve. Automated testing projects are often initiated only later to stumble, lose their way, and be thrown onto the ever-growing pile of failed projects.

Automation fails for many reasons—unachievable expectations is perhaps the most common, followed by inadequate allocation of resources (time, people, and money). Other factors include tools that are poorly matched to needs, the sheer impatience for success that hinders quality work, and a lack of understanding that automated testing is a different kind of software development, one that requires the same professional approach as all other development efforts.

Dorothy and Mark's previous book, *Software Test Automation: Effective Use of Test Execution Tools*, published in 1999, set the standard for books on this topic. The first part detailed practices found in most successful automation efforts—scripting techniques, automated comparison, testware architecture, and useful metrics. The second part described the experiences of a number of organizations as they implemented test automation efforts. Now, with an additional 10 years of industry knowledge behind them, Dorothy and Mark provide another set of organizational and personal experiences to guide our automation work. It brings us up to date, describing both the classical and most modern approaches to test automation. Each chapter tells a story of a unique automation effort—including both successes and failures—to give us guidance.

Certain themes reoccur in *Experiences in Test Automation*: reasonable and achievable objectives; management support; metrics, including return on investment; required skills; planning; setting expectations; building relationships; tools; training; and politics—all necessary to make test automation successful. However, these same themes are equally applicable at both the project and personal levels. One great benefit of this book comes from stepping outside the test automation realm and considering these themes in the larger context.

I first met Dorothy and Mark at the 1998 EuroStar conference in Munich. I was impressed with both their knowledge of and passion for helping others do great automated testing. I congratulate them for their outstanding accomplishment and commend this book to you.

—Lee Copeland
December 2011

Preface

Test automation tools have been around for about 30 years, yet many automation attempts fail, or at least are only partially successful. Why is this?

We wanted to understand if the principles of effective automation, as published in our previous book, *Software Test Automation*, are still relevant and what other principles now apply, so we began gathering information about real-world test automation implementations. This led us to a rather pleasant discovery: Over the past 10 years, many people have had good success with software test automation, many of them using our book. Of course, we are not the only ones to have described or discovered good automation practices, yet successful and lasting automation still seems to be an elusive achievement today. We hope the stories in this book will help many more people to succeed in their test automation efforts.

This book brings together contemporary automation stories. The technology of test automation has progressed significantly since our last book on automation was published in 1999. We wanted to find out what approaches have been successful, what types of applications are now being tested using test automation, and how test automation has changed in recent years. Different people have solved automation problems in different ways—we wanted to know what can be learned from their experiences and where and how test automation is being applied in new ways.

The case studies in this book show some approaches that were successful and some that were not. This book gives you the knowledge to help avoid the pitfalls and learn from the successes achieved in real life. We designed this book to help you get the most out of the real-life experiences of other professionals.

The case studies in this book cover mainly the automation of test execution, but other types of automation are mentioned in some chapters. We focus primarily on system-level automation (including user acceptance testing), although some chapters also cover unit or integration testing. Test automation is described for many types of applications, many environments and platforms; the chapters cover commercial, open source, and inhouse tools in traditional and agile development projects. We are surprised by the number of different tools being used—around 90 commercial and open source tools are listed in the Appendix (which includes any tools used by the chapter authors, not just testing tools).

The experiences described in this book are all true, even though in some cases the author or company name is not revealed. We encouraged the case study authors to describe what happened rather than offer general advice, so this book is very real!

In collecting this book's stories, we were struck by the pervasiveness of test automation into every industry and application. We were also impressed with the dedication and persistence of those who have developed test automation within their companies. Unfortunately, we were also struck by the difficulties that many of them encountered, which sometimes resulted in failure. We are sure the experiences described in this book can help you to be more successful with your test automation.

Case Studies Plus (Our Added Value)

This book is more than a collection of essays; we worked closely with the authors of the chapters to produce a book with information that we felt would be most useful to you. Our review process was thorough; we asked questions and suggested changes in several rounds of reviewing (special thanks are due to the chapter authors for their patience and additional information). Our "old versions" folder contains over 500 documents, so each chapter has been carefully crafted.

We help you get the most from this book by offering Good Points, Lessons, and Tips. Each chapter includes our own comments to highlight points we think should stand out at a glance. Watch for these helpful notes:

- **Good Points,** which are well worth noting (even if they are not necessarily new).

Good Point

Management support is critical, but expectations must be realistic.

- **Lessons,** often learned the hard way—things it would have been better not to do.

Lesson

Automation development requires the same discipline as software development.

- **Tips** on ways to solve particular problems in a way that seemed new or novel to us.

Tip

Use a "translation table" for things that may change, so the automation can use a standard constant term.

We picture these interjections as our way of looking over your shoulder as you read through the chapter and saying, "Pay attention here," "Look at this," and "This could be particularly useful."

How to Read This Book

Each case study is a standalone account, so the chapters can be read in any order. The arrangement of the chapters is designed to give you a variety of experiences if you do read the book from front to back.

To decide which chapter you would like to read first or read next, look at Table P.1, a "chapter selector" that summarizes characteristics of the various chapters. The table enables you to see at a glance which chapters cover a particular application, tool, development methodology, and so on, and helps you to quickly find the chapters most directly relevant to you. After Table P.1 are one-paragraph summaries of each case study chapter.

Following this Preface, the section titled "Reflections on the Case Studies" presents our overall perspective and summary of the management and technical issues discussed in the chapters along with our view and comments on those issues (and our diagram of testware architecture). In this section of the book, we distill the most important points of advice to those currently involved in, or about to embark on, their own automation. This is the "executive summary" of the book.

Chapters 1 to 28 are the case study chapters, each written by an author or authors describing their experience in their specific context: what they did, what worked well, what didn't, and what they learned. Some of the chapters include very specific information such as file structures and automation code; other chapters are more general. One chapter (10) is an update from a case study presented in *Software Test Automation*; the rest are new.

Chapter 29, "Test Automation Anecdotes," is a mini-book in its own right—a collection of short experience stories from over a dozen different people, ranging from half a page to several pages, all with useful and interesting points to make.

Finally, the Appendix, "Tools," covers the commercial and open source tools referred to in the chapters.

Table P.1 Case Study Characteristics

Chapter	Author	Application Domain	Location	Lifecycle	Number on the Project
1	Lisa Crispin	Financial, web	USA	Agile	9–12
2	Henri van de Scheur	Database	Norway		30–3
3	Ken Johnston, Felix Deschamps	Enterprise server	USA	Traditional with agile elements	>500
4	Bo Roop	Testing tool	USA	Waterfall	12–15
5	John Kent	Mainframe to web-based	UK	Traditional	40
6	Ane Clausen	2 projects: pensions and insurance	Denmark	None and agile	3–5
7	Elfriede Dustin	Government: Department of Defense	USA	Agile	100s
8	Alan Page	Device drivers	USA	Traditional	Hundreds
9	Stefan Mohacsi, Armin Beer	European Space Agency services	Austria, Italy, Germany	Traditional	>100
10	Simon Mills	Financial: insurance	UK	Chaotic and variable	Dozens
11	Jason Weden	Networking equipment	USA	Traditional (waterfall)	25

Time Span	Tool Type(s)	Pilot Study?	ROI Measured?	Successful?	Still Breathing?
1 yr, report after 6 yr	Open source	No	No	Yes	Yes
5–6 yr	Inhouse	No	No, but 2,400 times improved efficiency	Yes	Yes
~3 yr	Commercial, Inhouse	No	No	Yes	Yes
1 yr, 2 mo	Commercial	No	No	No	No
23 yr	Commercial	Yes	No	Yes	Yes
6 mo 1 yr	Commercial	No Yes	No Yes	No Yes	No Yes
4½ yr	Commercial, Open source, Inhouse	Yes	Yes	Yes	Yes
9 yr	Commercial, Inhouse	No	No	Yes	Yes
6+ yr	Commercial, Open source, Inhouse	No	Yes, projected payback after 4 cycles	Yes	Yes
15 yr	Commercial	No, but began small scale	No, but now running 5 million tests per month	Yes	Yes; client base still growing
3 yr	Inhouse	No	No	Ultimately, yes	Yes

Continues

Table P.1 Case Study Characteristics (Continued)

Chapter	Author	Application Domain	Location	Lifecycle	Number on the Project
12	Damon Yerg (pseudonym)	Government services	Australia	V model	Hundreds
13	Bryan Bakker	Medical devices	Netherlands	V model	50
14	Antti Jääskeläinen, Tommi Takala, Mika Katara	Smartphone applications in Android	Finland		2
15	Christoph Mecke, Melanie Reinwarth, Armin Gienger	ERP Systems (SAP), 2 projects: health care and banking	Germany, India	Traditional	10
16	Björn Boisschot	SAP applications in the energy sector	Belgium	Traditional	12
17	Michael Williamson	Web-based, distributed	USA	Agile	15
18	Lars Wahlberg	Financial marketplace systems	Sweden	Incremental to agile	20 (typical)
19	Jonathan Kohl	Various, web to embedded	Canada	Agile and traditional	A few –60

Time Span	Tool Type(s)	Pilot Study?	ROI Measured?	Successful?	Still Breathing?
11 yr	Inhouse	Yes	No, but comparable manual effort calculated	Yes (peaks and troughs)	Yes; thriving and forging ahead
1.5 yr	Commercial, Open source, Inhouse	Started small	Yes	Yes	Yes
6–8 mo	Commercial, Open source	Entire project is a pilot study	No	Yes	Yes
4 yr 2 yr	Commercial, Inhouse	No	No	Yes	Yes
6 mo	Commercial	Yes	No	Yes	Yes
6 mo	Commercial, Open source	Yes	No	No	No
~10 yr	Open source	Yes	Yes, projected payback for tests run daily, weekly, or monthly	Yes	Yes
Various	Commercial, Open source, Inhouse	Yes, in some cases	No	Yes	Yes; some still in use

Continues

Table P.1 Case Study Characteristics (Continued)

Chapter	Author	Application Domain	Location	Lifecycle	Number on the Project
20	Albert Farré Benet, Christian Ekiza Lujua, Helena Soldevila Grau, Manel Moreno Jáimez, Fernando Monferrer Pérez, Celestina Bianco	4 projects, all medical software	Spain, USA, Italy	Spiral, prototyping, waterfall	2–17
21	Seretta Gamba	Insurance	Germany	Iterative	27
22	Wim Demey	Customized software packages	Belgium	Traditional V model	
23	Ursula Friede	Insurance	Germany	Traditional (V model)	30
24	John Fodeh	Medical applications and devices	Denmark	Traditional (V model), incremental	30
25	Mike Baxter, Nick Flynn, Christopher Wills, Michael Smith	Air traffic control	UK	Traditional	15–20
26	Ross Timmerman, Joseph Stewart	Embedded: automotive systems	USA	Phased waterfall	8

Time Span	Tool Type(s)	Pilot Study?	ROI Measured?	Successful?	Still Breathing?
5 yr 2 yr Few months 1 yr	Commerical Inhouse Commercial Commercial	No	No	Yes Partly No Yes	Yes Yes No Planned
12 mo	Commercial, Inhouse	Yes	No	Yes	Yes
4 mo	Commercial, Open source	Yes	No	Yes	Yes
~6 mo	Commercial	No	No, but quantified savings of €120,000 per release	Yes	Yes
6 yr	Commercial, Inhouse	Yes	No	Yes	Yes
Cycles lasting 3–12 mo	Commercial, Open source, Inhouse	Yes	No	Yes	Yes
5 yr	Inhouse with commercial hardware	No	No	Yes	Yes

Continues

Table P.1 Case Study Characteristics (Continued)

Chapter	Author	Application Domain	Location	Lifecycle	Number on the Project
27	Ed Allen, Brian Newman	Web-based, mobile, desktop, social channels (voice, chat, email)	USA	Traditional	28
28	Harry Robinson, Ann Gustafson Robinson	Problem reporting for telephone systems	USA	Waterfall	30 overall 4 on project

Chapter Summaries

Chapter 1, An Agile Team's Test Automation Journey: The First Year

Lisa Crispin describes, in her very engaging style, what happened when an agile team decided to automate their testing. Given Lisa's expertise in agile, you will not be surprised to see that this team really was agile in practice. One of the interesting things about this project is that everyone on the team (which was fairly small) was involved in the automation. Not only did they excel in agile development, they also developed the automation in an agile way—and they succeeded. Agile development was not the only component of this team's success; other factors were equally important, including building a solid relationship with management through excellent communication and building the automation to help support creative manual testing. Another key factor was the team's decision to build in process improvement along the way, including scheduling automation refactoring sprints twice a year. You are sure to agree that what Lisa and her team accomplished in their first year is remarkable. The project was done for a United States company in the finance sector.

Time Span	Tool Type(s)	Pilot Study?	ROI Measured?	Successful?	Still Breathing?
1 yr	Commercial, Inhouse	No	No, but benefits measured	Yes	Yes
1.5 yr	Inhouse	No	No	Yes	No

Chapter 2, The Ultimate Database Automation

Henri van de Scheur tells a story that spans half a dozen years, relating what happened when he and his colleagues developed a tool for testing databases in multiple environments. They set good objectives for their automation and a good architecture for the tool. They automated so many tests that they developed a lifecycle for automated tests that included periodic weeding. Tests were run nightly, weekly, or with special scheduling. Despite great success, a number of problems were encountered, and Henri describes them honestly. The development of this database testing tool (now open source) was done in Norway by a small team, over several years, and it achieved a very impressive return on investment.

Chapter 3, Moving to the Cloud: The Evolution of TiP, Continuous Regression Testing in Production

Ken Johnston and Felix Deschamps from Microsoft describe how they moved from product-based to service-based automated testing by implementing the automation in the cloud. Testing of Microsoft Exchange servers was already extensively automated, and much of the existing automation was reusable. Testing in production seems a foreign concept to most testers, but this chapter explains why it was necessary and beneficial to move to continuous monitoring and contains useful tips for anyone considering a similar move. This experience takes place in the United States, over three years, and unsurprisingly, Microsoft tools were used.

Chapter 4, The Automator Becomes the Automated

Bo Roop takes us on a guided tour of attempting to automate the testing of a test automation tool. One of the first questions to ask a tool vendor is "Do you test the tool using the tool?" But the answer isn't as straightforward as you might think! With his lively writing style, Bo gives an honest description of the difficulties and challenges he encountered, particularly in the verification of test results. It is a good idea to find out what others have tried, and Bo shows the advantages of doing so. His sensible approach to automation is to start by automating the easier components before tackling the more complex. Unfortunately, this story does not have a happy ending. It illustrates how presumably well-intentioned management actions can sabotage an automation effort. For reasons that become obvious when you read this chapter, the tool vendor is not identified: a fictitious company and tool name are used instead. This experience takes place in the United States with one automator (the author) and covers just over one year.

Chapter 5, Autobiography of an Automator: From Mainframe to Framework Automation

John Kent tells us how and when test automation started and offers surprising information about the origins of capture/replay technology. Understanding how automation worked on mainframes shows how some of the prevailing problems with test automation have developed; approaches that worked well in that environment did not work well with GUIs and the need to synchronize the test scripts with the software under test. The principles John discovered and put into practice, such as good error handling and reporting and the importance of testing the automation itself, are still relevant and applicable today. John's explanation of the economic benefits of wrappers and levels of abstraction are compelling. He ends with some recent problem/solution examples of how web elements can trip up the automation. This United Kingdom–based project involved mainly commercial tools.

Chapter 6, Project 1: Failure!, Project 2: Success!

Ane Clausen tells of two experiences with test automation, the first one unsuccessful and the second one a solid success, largely due to what she learned from her first experience. Lessons are not always so well learned—which is a lesson in itself for everyone! Ane's first story is told honestly and highlights the serious impact of insufficient management support and the importance of choosing the right area to automate. In her second story, Ane designed a three-month pilot study with clear objectives and a good plan for achieving them. Many useful lessons are described in this chapter, such as good communication (including using the walls), limited scope of the early automation efforts, good use of standards in the automation, a good structure (looking for common elements), and keeping things simple. The continuing automation was then built on the established foundation. Ane's experience was with pension and insurance applications in Denmark, using commercial tools.

Chapter 7, Automating the Testing of Complex Government Systems

Elfriede Dustin, well known in the test automation world, shares her experience of developing an automation framework for real-time, mission-critical systems for the U.S. Department of Defense. Because of the particular type of software that was being tested, there were specific requirements for a tool solution, and Elfriede and her colleagues needed to spend some time searching for and experimenting with different tools. Their clear statement of requirements kept them on track for a successful outcome, and their eventual solution used a mixture of commercial, open source, and inhouse tools. They met with some unexpected resistance to what was technically a very good system. This story covers hundreds of testers and tens of automators, testing millions of lines of code, over a period of four and a half years.

Chapter 8, Device Simulation Framework

Alan Page from Microsoft tells a story of discovery: how to automate hardware device testing. We all take for granted that our USB devices will work with our computers, but the number of different devices that need to be tested is very large and growing, and it was difficult to automate such actions as unplugging a device. However, a simulation framework was developed that has enabled much of this testing to be automated in a way that has found widespread use inside and outside of Microsoft. The chapter includes numerous examples showing both the problems encountered and the solutions implemented. This story is from the United States and was an inhouse development now used by hundreds of testers.

Chapter 9, Model-Based Test-Case Generation in ESA Projects

Stefan Mohacsi and Armin Beer describe their experience in using model-based testing (MBT) for the European Space Agency (ESA). Their team developed a test automation framework that took significant effort to set up but eventually was able to generate automated tests very quickly when the application changed. This chapter includes an excellent return-on-investment calculation applicable to other automation efforts (not just MBT). The team estimated break-even at four iterations/releases. The need for levels of abstraction in the testware architecture is well described. The application being tested was ESA's Multi-Mission User Information Services. The multinational team met the challenges of automation in a large, complex system with strict quality requirements (including maintainability and traceability) in a waterfall development—yes, it can work! If you are thinking of using MBT, you will find much useful advice in this chapter. A mixture of inhouse, commercial, and open source tools were used by the team.

Chapter 10, Ten Years On and Still Going

Simon Mills updates his case study from our previous book, *Software Test Automation* (Addison-Wesley, 1999). Still automating 10 years on is a significant achievement! The original story is included in full and contains excellent lessons and ideas. The success and continued growth of this automation is a testament to the sound foundation on which it was built more than a decade ago. The case study describes many lessons learned the hard way and some amusing observations on Simon and his team's first automation attempts. Their automation architecture separated their tests from the specific tools they were using—a wise move as was proved later. They devised a reliable way to document their tests that has stood the test of time. This story takes place in the United Kingdom, uses commercial tools, and covers about 15 years.

Chapter 11, A Rising Phoenix from the Ashes

Jason Weden tells a story of initial failure leading to later success. The failure of the first attempt at automation was not due to technical issues—the approach was sound. However, it was a grassroots effort and was too dependent on its originator. When he left, the automation fell into disuse. But the phoenix did rise from the ashes, thanks to Jason and others who had the wisdom to build on what had gone before, making many improvements to ensure that it was more widely used by business users as well as technical people. Their "test selector" for choosing which tests to execute gave the test engineers flexibility, and they ensured their legitimacy by keeping stakeholders informed about bugs found by automated tests. The small team that implemented automated testing for home networking equipment is based in the United States.

Chapter 12, Automating the Wheels of Bureaucracy

Damon Yerg (a pseudonym) tells of experiences in automating large systems for a government agency, over more than 10 years, with hundreds of developers and testers and more than a dozen automators. After some uncertain starts, external pressure brought the right support to move the automation in the right way. The tests to be automated covered diverse applications from web-based to mainframes, all with environmental factors. This story brings home the need for automation standards when many people are using the automation. Damon and his colleagues organized the regression library into core and targeted tests to enable them to be selective about which tests to run, and they related the automated tests to risk factors. The basic design of the automation supported business expert testers and offered technical support as needed. One of the most powerful things they did to ensure continuing management support was to develop a spreadsheet describing the benefits in a way that communicated effectively to stakeholders. This is a very successful large-scale automation project from Australia.

Chapter 13, Automated Reliability Testing Using Hardware Interfaces

Bryan Bakker tells of automating testing for medical devices, an area with stringent quality requirements and difficulties in accessing the software embedded in devices such as X-ray machines. Bakker and his team's starting point was simple tests that assessed reliability; functional testing came later. The automation was developed in increments of increasing functionality. The chapter contains many interesting observations about management's changing views toward the automation (e.g., management was surprised when the testers found a lot of new bugs, even though "finding bugs" is what management expected of them). The team developed a good abstraction layer, interfacing through the hardware, and were even able to detect hardware issues such as the machines overheating. The results in the test logs were analyzed with inhouse tools. The reliability testing paid for itself with the first bug it prevented from being released—and, in all, 10 bugs were discovered. Subsequent functional testing was smoother, resulting in cutting the number of test cycles from 15 to 5. This story is from the Netherlands, and the project had excellent success using commercial and inhouse tools with just two people as the test automators.

Chapter 14, Model-Based GUI Testing of Android Applications

Antti Jääskeläinen, Tommi Takala, and Mika Katara tell how they used model-based testing (MBT) in a substantial pilot study, testing smartphone applications—specifically Messaging, Contacts, and Calendar—on the Android platform. They used domain-specific tools rather than generic testing tools and a mix of commercial, open source, and inhouse tools, including two MBT tools. Testing in a huge state space was quite a challenge, but they give clear examples of the types of tests they were automating and good explanations of the algorithms used. They include many helpful ideas for making testing easier, such as using correct syntax for keywords and converting testware to more usable formats. This story covers a pilot project in Finland.

Chapter 15, Test Automation of SAP Business Processes

Christoph Mecke, Melanie Reinwarth, and Armin Gienger tell how automation is used in testing major application areas of SAP, specifically banking and health care. Because SAP applications are deployed in many client sites and have a long life, the test automation is on a large scale, with over 3 million automation lines of code and 2,400 users of the automation. The testware architecture of the tool they developed is modular. The standards and guidelines put in place ensure the automation can be used in many areas and in many countries, and the tool can be used by SAP customers as well. The automated tests must be ready to go as soon as new software is delivered to enable the testing to help speed delivery rather than slow it down. Some of the problems they encountered concerned testing parallel versions and multiple releases, consistency of test data environments, setting of customized parameters, and country-specific legal issues. One particularly good idea the authors relate is to have a tool do static analysis on the test automation scripts. They also warn about ending up with "zombie scripts": dead automation code in a script. This story takes place in Germany and India over several years.

Chapter 16, Test Automation of a SAP Implementation

Björn Boisschot tells how he developed a generic framework based on his observations while setting up automation for various clients. He explains how he used the framework and extended it in the automation of the testing for two SAP applications in the energy sector. The groundwork for the project is well laid, with a proof-of-concept forming the basis of the go/no-go decision. Björn gives good examples of communicating with managers, explains why capture/playback does not work, and emphasizes the importance of setting realistic expectations. The framework, now a commercial product, used good automation and software development principles to construct a modular structure that successfully scaled up. The layers of abstraction are well implemented, separating the automated tests from the tool and giving users access to tests without having to program. He ends by showing how multilingual tests were implemented through a translation table and some of the challenges of that project. This case study takes place in Belgium and is the origin of a commercial framework.

Chapter 17, Choosing the Wrong Tool

Michael Williamson tells the story of trying to use a tool (which he inherited) that he later realized was not suitable for what he was trying to test. He was trying to automate the testing of major functionality in web development tools at Google, where he was new to testing and test automation. Some assumptions that seemed obvious at first turned out to be not as they seemed. His approach to taking over the use of an existing tool seemed sensible, yet he encountered a lot of problems (which are illustrated), particularly in automated comparison. Michael found it was difficult to abandon something you have put a lot of effort into already, yet in the end, this was the best approach (and he was surprised when he discovered what had really been hindering his efforts). This story is of one person in the United States attempting to use a commercial tool.

Chapter 18, Automated Tests for Marketplace Systems: Ten Years and Three Frameworks

Lars Wahlberg gives an insight into his 10 years of test automation for marketplace systems, including the development of three automation frameworks. One of the key messages in this chapter is the importance of having the right abstraction level between the technical aspects of the automated tests and the testers. Lars describes how they addressed this issue in each of the different frameworks and some of the problems that occurred if the abstraction layer was too thick or too thin. As Lars and his team moved into agile development, they found the process worked best when they had people who could be both tester and test automator, but the role of the test automation architect was critical for smooth implementation of automation. The chapter illustrates the progression of a test from manual to automated and the automated checking of preconditions. Lars also includes an illuminating assessment of return on investment for automation based on how often tests are run, the costs of automating, and the number of tests that are automated. This work was done in Sweden using open source tools.

Chapter 19, There's More to Automation than Regression Testing: Thinking Outside the Box

Jonathan Kohl takes us through a set of short stories, each illustrating a variation on the theme of automating things other than what people usually think of, that is, thinking outside the box. The stories together demonstrate the value of applying ingenuity and creativity to solve problems by automating simple tasks, or processes other than test execution. Full-scale automation is not necessarily a practical option in some cases, but small and simple tool solutions can provide significant benefits and savings. Jonathan shows that even devising "disposable" automation scripts can be very useful. These experiences from Canada cover small to large applications, from web-based to embedded systems, using commercial, open source, and inhouse tools.

Chapter 20, Software for Medical Devices and Our Need for Good Software Test Automation

Even if you are not working with medical devices, this chapter, written by Albert Farré Benet, Christian Ekiza Lujua, Helena Soldevila Grau, Manel Moreno Jáimez, Fernando Monferrer Pérez, and Celestina Bianco, has many interesting lessons for anyone in automation. For the safety-related applications they test, a strict, formal test process is used with a risk-based approach to the implementation of the automation. Their story shows how success can be mixed even across different projects in the same organization; attempts to resurrect previous automated tests that had fallen into disuse resulted in different outcomes in different projects. In some cases, unrealistic imposed objectives ("total automation") virtually guaranteed failure. However, they did progress, devising a good list of what they wanted to use automation for and what they wanted to avoid automating. The problems encountered included some that are unique to regulated industries (where test tools need to be formally validated before their results can be used in the qualification testing of the software), problems with hardware interaction, and problems with custom controls (overcome by an abstraction layer). Their honest description of problems, solutions, and lessons learned in the four projects described is as useful as it is interesting. These stories from Spain involve commercial and inhouse-developed tools and cover projects lasting a few months to five years.

Chapter 21, Automation through the Back Door (by Supporting Manual Testing)

Seretta Gamba tells of the difficulties she and her team experienced in trying to progress automation, even though it had a good technical foundation. She shows how they addressed the real needs of the testers and provided support where testers needed it most, which was in manual testing. The really interesting twist is how they were able at the same time to harvest things that would progress the automation—win-win situations. Their approach is described in detail, showing how they implemented levels of abstraction, separating tests from application detail such as the GUI, and they include an example of a user interface (UI) to the tests. They developed what they call command-driven testing, which is based on a form of keyword-driven testing and worked well for them. This case study takes place in the insurance industry in Germany.

Chapter 22, Test Automation as an Approach to Adding Value to Portability Testing

Wim Demey describes how he developed an inhouse tool to work with commercial and open source tools to enable parallel testing of different configurations. Testing the installation of packages that are developed with a common core but customized for different customers is important because failures in installation may have serious impacts not just on the system but on market perception as well. But testing the ports of the different packages on a large variety of configurations (operating systems, browsers, databases) can be a time-consuming and resource-hungry process, so it is a good candidate for automation. Wim used virtualization to run portability tests on many configurations at once through a custom-built tool that launches the portability tests in the virtual environments. The chapter offers many good lessons, including the importance of automating the activities surrounding test execution, for those considering a similar task. This story comes from Belgium and involves financial client software.

Chapter 23, Automated Testing in an Insurance Company: Feeling Our Way

Ursula Friede describes the experience of "feeling their way" to better automation. She and her team did not plan the steps they ended up taking, but it would have been a good plan if they had. They began by just experimenting with a tool but soon realized its limitations, so they decided to focus on the most pressing problems by changing their automation. In each of four phases, they successfully addressed an existing problem but then encountered new problems. They calculated an impressive saving per release after they had implemented their improvements. Ursula was based in Germany at the time and was automating tests in the insurance sector.

Chapter 24, Adventures with Test Monkeys

John Fodeh tells of his experiences with automated random test execution, also known as "monkey testing," for medical devices used for diagnosis and in therapeutic procedures. John describes the limitations of automated regression testing and why few bugs are found that way. Using test monkeys enabled him to find many bugs in the devices that otherwise would not have been found before release. Inputs are selected from a list at random, and all crashes are investigated. Test monkeys can be implemented with commercial execution tools or inhouse tools. This approach borders on model-based testing and also is close to exploratory test automation. John's team was able to calculate the reliability of the devices over time, which helped in the release decision. Many other interesting metrics are described in this chapter as well. An honest assessment of the limitations of the technique makes this a well-balanced description of an interesting use of tool support in testing. This story takes place in Denmark.

Chapter 25, System-of-Systems Test Automation at NATS

Mike Baxter, Nick Flynn, Christopher Wills, and Michael Smith describe the automation of testing for NATS (formerly called National Air Traffic Services Ltd.). Among its other responsibilities, NATS provides air traffic control services for the airspace over the eastern North Atlantic Ocean. Testing a safety-critical system where lives are at stake requires a careful approach, and the requirements included special technical factors, human factors, and commercial considerations. The tool used was able to test the software without affecting it, and it was also useful in some unexpected ways, such as in training. The authors provide a helpful checklist for deciding which tests to automate and describe lessons learned, both general and technical. This case study is from the United Kingdom and involves commercial, open source, and inhouse tools.

Chapter 26, Automating Automotive Electronics Testing

Ross Timmerman and Joseph Stewart tell about the inhouse testing tools they developed over the years to test automotive electronics systems at Johnson Controls. In 2000, no commercial tools were available for this niche area, so the Johnson Controls team chose to develop their own tools, which was done initially by the test team and later by an offshore team. Both benefits and problems arise from writing your own tools, but the team dramatically increased the number of tests they were able to run. They also discovered ways to automate things they initially thought had to be done manually. In their second initiative, they built on the success of the earlier tools but provided more functionality, using off-the-shelf hardware modules with their own software to get the best results. This case study covers five years and is based in the United States.

Chapter 27, BHAGs, Change, and Test Transformation

Ed Allen and Brian Newman tell of their experience in automating the testing for a customer relationship management system. The problems they encountered ranged from technical issues, such as environments, testware architecture/framework, and "script creep," to people issues, including management support, working with developers, and when to ignore a consultant. After several false starts, they were challenged by some "Big Hairy Audacious Goals" (BHAGs) and were given support to meet them. They achieved good benefits in the end and provide some intriguing metrics about the number of defects found by different ways of testing (automated, manual scripted, exploratory testing, and bug fixes). This story is based in the United States with a team of 28, including 25 from quality assurance and 3 from development.

Chapter 28, Exploratory Test Automation: An Example Ahead of Its Time

Harry Robinson and Ann Gustafson Robinson describe Harry's experiences in doing what seems on the surface like an oxymoron. How can testing be both exploratory and automated? There are a number of requirements to make it possible, but when it can be done, it provides a way to explore test input spaces that are far too large to be tested in any other way. This chapter takes us step by step through Harry's journey of implementing what has now become known as model-based testing. The effort was far more successful than anyone thought possible at the time, which had an unexpected side effect on Harry's career! Although this story takes place rather a long time ago, the techniques described are now "coming into their own" because better tool support makes them applicable to more systems in a wider context. (Note: Although this chapter has two authors, it is told from Harry's viewpoint for easier reading.) This story comes from the United States.

Contributors and Acknowledgments

Submissions for inclusion in this book came from people we knew, people we met at conferences, and those who responded to the appeal for authors on our websites and in our newsletters. We began work on the book in autumn 2009, and from December 2009 until December 2011, the two of us put in over 850 hours on this book; this does not include the effort from all of the chapter and anecdote authors.

Thanks to all of the chapter authors for their tremendous efforts, more often than not on their own time, to produce their respective chapters and liaise with us through the production of this book. Thanks to the authors who contributed to Chapter 29: Michael Albrecht, Mike Bartley, Tessa Benzie, Jon Hagar, Julian Harty, Douglas Hoffman, Jeffrey S. Miller, Molly Mahai, Randy Rice, Kai Sann, Adrian Smith, Michael Snyman, George Wilkinson, and Jonathon Lee Wright.

Thanks to a number of other people who helped with this book, including Claudia Badell, Alex Blair, Hans Buwalda, Isabel Evans, Beth Galt, Mieke Gevers, Martin Gijsen, Daniel Gouveia, Linda Hayes, Dion Johnson, Audrey Leng, Kev Milne, Andrew Pollner, David Trent, and Erik Van Veenendaal. Thanks to authors for co-reviewing other chapters.

Thanks to the Pearson publishing team for their help, encouragement, and all the work they have done to make this book a reality: Christopher Guzikowski, Raina Chrobak, Sheri Cain, Olivia Basegio, and freelancer Carol Lallier.

—Dorothy Graham
Macclesfield, United Kingdom
www.dorothygraham.co.uk

—Mark Fewster
Llandeilo, United Kingdom
www.grove.co.uk

December 2011

CHAPTER 29

TEST AUTOMATION ANECDOTES

An anecdote is a short account of an incident (especially a biographical one). Numerous people told us short stories (anecdotes) of their experiences, and because they merit retelling but don't constitute full chapters, we collected them in this chapter. The stories vary in length from half a page to five pages. They are all independent, so they can be read in any order.

29.1 Three Grains of Rice

Randy Rice, United States
Consultant, speaker, and author

As a consultant, I see a number of different situations. The following describes three short experiences I have had with a couple of clients.

29.1.1 Testware Reviews

I was a consultant once on a project where we were trying to bring best practices in test automation into a large organization that had only tinkered with test automation. The company's environment spanned web-based, client/server, and mainframe applications. About 15 test designers and 15 test automators were brought in to work on this effort. The test tools in use when we first arrived were old versions not even being supported by the vendor because of their age. Only a small portion of the applications were automated to any degree. The automation that was in place consisted of large test scripts that were very difficult to maintain.

The project was initiated as one of several aggressive projects to technically reengineer the entire IT operation. The chief information officer (CIO) who was the original champion of these projects was a believer in test automation. Her successor inherited the projects but did not share the same commitment and enthusiasm for many of them. There was also a 6-month vacancy while the new CIO was being recruited, so things had just coasted along. When the new sheriff came to town, people started trying to figure out who would survive.

Supervising this effort were three senior test automation consultants who really knew their stuff and had a very specific approach to be followed. We had six test automation gurus on the managing consultant side, and we had regular communication based on metrics and goals. In fact, we developed a very nice dashboard that integrated directly with the tools. At any time on any project, people could see the progress being made. We gave demonstrations of how the automation was being created (this went over management's heads) and also the results of automation, so we had plenty of knowledge and communication.

To their credit, the contracting company trained all the incoming test design and automation consultants out of their own pocket. Although these were experienced consultants, the contractor wanted to set a level baseline of knowledge for how the work would be done on this project.

After about 3 weeks, it became apparent that some of the test automators were going their own way and deviating from the defined approach. This was a big problem because a keyword approach was being used, and certain keywords had to work consistently among applications. There were too many people who wanted to do things their way instead of the way that had been designed.

To correct the issue, the senior consultants required all test designers and consultants to attend daily technical reviews of testware. Technical reviews are not just for application software code or requirements. To get 30 people (more or less) from diverse backgrounds on the same approach is not a trivial achievement! Before long, this became a peer review type of effort, with critiques coming from peers instead of the senior consultants. It had turned into a forum for collaboration and learning.

Good Point

Reviews of automation testware are beneficial not just from a technical standpoint but also from an idea-sharing and brainstorming standpoint.

Some of the test consultants resisted the technical reviews and didn't last on the project. They were the same test automators who refused to follow the designed approach.

After a few weeks, it was no longer necessary to maintain the frequent reviews, and the test automation effort went a lot more smoothly.

Unfortunately, test management and senior technical management (at the CIO level) in this organization never saw the value of test automation. Therefore, much of the fine work done by this team was scrapped when senior management pulled all support for this effort. They terminated the contracts of everyone who knew anything about the automation and ended up "achieving" a negative return on investment (ROI)—millions of dollars were spent with very little to show for it. I see little future for automation at this company now, in spite of the great work that was done.

Lesson

Technical success is no guarantee of lasting automation; management support through good communication is needed.

This was a huge and very visible project. But the test manager was like many test managers and had been thrust into the role with no training in testing. The client staff were thin in numbers, skills, and motivation.

My bottom line assessment is that the organization simply was not ready for such an aggressive project. Then, when the sponsoring CIO left, there was no one to champion the project. Also, the software wasn't engineered in a way that was easily automated; it was old and very fragile. The expectations for ROI were very high and it would have been better to take smaller steps first.

29.1.2 Missing Maintenance

There was a move in the late 1990s to go from fractional stock prices to decimal prices. For decades, stock prices had been shown as "$10 1/2" instead of "$10.50." There were many benefits to the decimal representation, such as ease of computation, standardization worldwide, and so forth. This was a major conversion effort that was almost as significant for the company as the Y2K maintenance effort.

Because the conversion effort was so massive and time was so short, management decided not to update the test automation during the project. This decision later proved to be significant.

By the time the decimalization project was complete, work was well underway for the Y2K conversion effort. We wanted to update the test automation for both efforts—decimalization and Y2K—at the same time. However, the schedule won again, and by the time the Y2K effort was complete, the test automation was deemed to be so out of date, it would be easier to start all over in a new, more modern tool. This was indeed the case. One of the problems was the platform, the DEC VAX. There was only one tool on the market for that platform. An emulator-based PC tool could have been used, but then there would be issues of character-based testing.

At the time, keyword-driven or even data-driven approaches were not widely known, and the automators and test managers encountered for themselves the difficulties of maintaining the automation code with hardcoded values. The first decision not to keep up with maintenance of the automated testware proved to be the death of the entire test automation effort for that application. This was a highly complex financial application, taking about 3 years to create the original test automation. There were new projects being developed on client/server platforms. Starting again from square one might have been a good idea, but the company hadn't yet realized the ROI from the first effort. Basically, the manual test approach just seemed too compelling.

Lesson

Once you abandon the maintenance of the automation, it is likely to die. For a better chance of success, choose an automation approach that will require the least maintenance.

29.1.3 A Wildly Successful Proof-of-Concept

I was hired by a large Wall Street company to assess the quality of its software testing process and make recommendations regarding a workable test automation approach. This company was not new to the idea of test automation. In fact, it already had three major test automation tools in place and was looking for another test automation solution. There was no integration between the various test automation tools, and they were applied in functional silos.

One particular system at this company was being manually regression tested every day! This one very unfortunate lady performed the same tests every day for about 8 hours a day.

As we were considering which tools might be the best fit for this system, I suggested that we contact the various candidate vendors and see if any were willing to send a technical consultant and perform a proof-of-concept using the the vendor's tool and my client's system.

My client thought this was an excellent idea, so we contacted the vendors and found one that was willing to send in a consultant at a reduced daily rate. We felt it was worth the risk to pay for the proof-of-concept. It would have taken us weeks to try to get an unfamiliar tool working, and we didn't want to pay for a tool without knowing it would work.

It seemed to me a good test project for the proof-of-concept was the 8-hour daily manual regression test, so we asked the vendor's test automation consultant to tackle that application.

After 3 days, the regression tests were completely automated! We were hoping just to get an idea that the tool would work in the environment. What we got instead was our first success! We probably broke even on ROI after 1 month.

Tip

Tool vendors can be a great help to get you started "on the right foot."

My client was thrilled, I looked good for suggesting the idea, and the vendor made a big sale. However, the person happiest with the outcome was the lady who had previously performed the manual regression tests for 8 hours a day. Now, she started an automated test and 15 minutes later, the test was done. Her time was now free for designing better tests and performing more creative tests.

29.2 Understanding Has to Grow

Molly Mahai, United States
Test manager

When we started looking at automation, I read everything I could get my hands on. I knew I couldn't automate everything. I knew automation would not replace people, and I knew it would require investment. I read *Software Test Automation* (Fewster and Graham, Addison-Wesley, 1999) at a recommendation, and I learned about the steps to take.

The funny thing is that even knowing all that, I felt I could not accomplish the first steps to set up an architectural framework. I know that sounds simple, but what does an architecture look like? How do we want to capture our tests? How will we set it up so that we can reuse scripts? All these questions kept preventing us from making progress. So, in my usual fashion, I made a decision and picked an architecture. I had no idea if it would work or not, but we needed to get moving with something. This freed us up to learn, and learn we did. We created a regression suite that addressed a handful of our applications, and it looked like we were moving in the right direction, but we ran into problems. There were too many scripts, and the initial grouping (directory structure) was not sufficient for our use.

This time, though, I knew a lot more and figured out that our architecture was lacking. We had too many projects, the library was cumbersome, and so on. So, I redesigned the architecture and created staging areas, including a sandbox area for development scripts, a place for scripts in use, and a place for scripts that were part of the production suite. We also enforced the naming conventions that we had put in place. These simple changes fixed a good many of our organizational problems.

The key is that we knew about this potential pitfall, and we knew how important it was to have an architecture that worked for us, but we couldn't design that architecture until we knew more of what we wanted to accomplish. For us, this was not a pitfall that we could avoid: We had to learn our way through it. The great thing was that I was intently aware of this potential problem (from reading the book), and I kept my eye on it. We redesigned the architecture as soon as we realized it wasn't working for us, and the impact on our automation effort was negligible.

I relate this to trying to explain to teenagers that they will view things differently when they are older. They cannot grasp it through hearing from someone else; they must learn it for themselves.

Lesson

Experiencing the problem is often the best (or only) way to see a better solution.

29.3 First Day Automated Testing

Jonathon Lee Wright, United Kingdom
Consultant and speaker

In the past decade, I have dealt with a number of ways of creating testware frameworks and found advantages and disadvantages in each of the following approaches:

- Modularity driven
- Keyword driven
- Data driven

In 2009, I was tasked with automating the testing of a new package for the New Zealand Lotteries Commission. The requirements phase had only just been completed, and the scheduled testing was looking very tight—less than a month (the schedule was imposed on us by marketing, who had already overcommitted their advertising!).

With just 2 months until the release would be delivered, this was an opportunity to adapt the testware frameworks I had used before, combining them with the latest technologies to develop what I refer to as a hybrid (keyword-/data-driven) automation framework.

Not only would this meet all the requirements set by the business, but more important, it would allow us to start the development of the testware framework immediately.

29.3.1 Initial Investment

The inherent problem with the hybrid approach is the large initial investment in time and effort to design the testware framework. Consequently, it was important that the development of this framework be treated as a legitimate project in its own right with its own planning, design, development, and testing phases together with the necessary resources. Table 29.1 shows our estimates for the effort required to create 100 test scenarios.

In essence, the hybrid approach would take roughly twice as long as the previous automation approach, which in turn would take twice as long to automate than the preparation for manual testing.

29.3.2 What Is to Be Automated?

Given the limited time available and the increased initial effort required, it was critical that we identified the optimum test coverage. To avoid developing unnecessary test components, we used the MoSCoW method:

- What *must* be automated?
- What *should* be automated?
- What *could* be automated?
- What *won't* be automated?

Table 29.1 Simple Calculations for Initial Testware Preparation Effort

Approach	Effort
Manual	2 weeks
Existing framework	1 month
Hybrid framework	2 months

This method allowed us to focus our efforts on those test components that were necessary and would give the greatest value by assessing the associated business risk, reusability, usage, complexity, and feasibility of each test component.

The test components were regarded as individual jigsaw pieces, but we kept in mind what was needed to complete the entire puzzle.

Business process modeling (BPM) was used within the centralized test management tool (Quality Centre) to represent the individual puzzle pieces (test components); mapping the pieces revealed the workflow through the system.

Figure 29.1 shows how one BPM may only include 5 puzzle pieces but enable more than 20 different paths through the system under test (SUT), with each path having a different business risk and impact assessment.

This made it easier to decide which automation modules to develop first by starting small and automating only the most critical business processes—keeping it as simple as possible while recognizing that the test coverage could be increased as the framework matured over time.

The decomposition of the workflows into a high-level model visualizes and enables an agile approach to the framework development. The development's build order and resource focus becomes obvious.

Each path through the system represents an agile feature, which may be in or out of scope depending on time and resources. Another benefit of this approach is that the model becomes an artifact that may be shared between the test framework and target application developers.

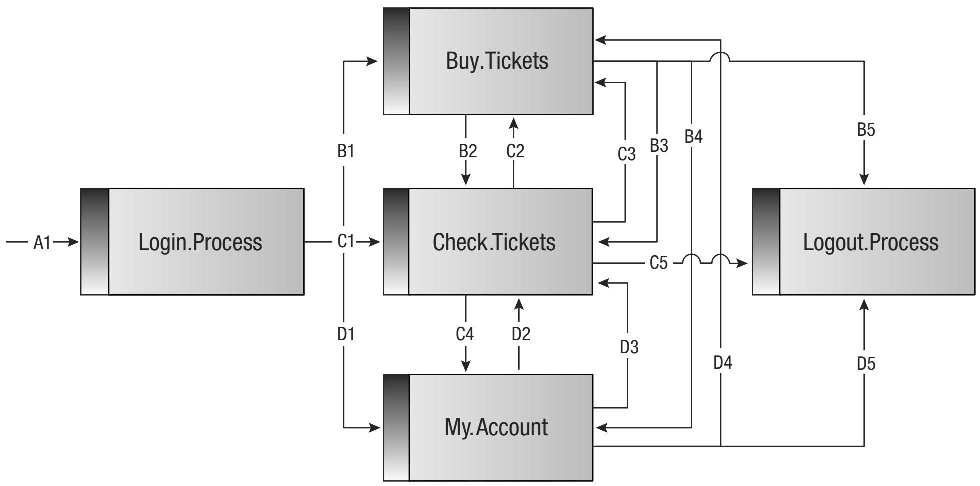

FIGURE 29.1 Business process model of the SUT.

Good Point

Start by automating the most valuable tests, but plan for later growth.

29.3.3 First Day Automated Testing

The key to first day automated testing is to create a dynamic object repository based on a combination of fuzzy logic and descriptive programming supporting the design and development of test scenarios before the actual delivery of the SUT.

Traditionally, because of the dependency on building the object repository, test automation is carried out at the end of the software development lifecycle once the SUT is delivered. However, because we had only a single month in which to execute testing but a full 2 months before the SUT was delivered, it seemed logical to develop the testware framework beforehand while the application was still in development.

Good Point

Automation can (and should) start before the software being tested is delivered, so the automated tests are ready to run on the first day the software is released. But this requires good planning and good testware architecture.

29.3.3.1 Business-Level Keywords

To allow the creation of test scenarios ahead of the SUT delivery, a high-level keyword approach was used to represent:

- Specific BPM and business process testing (BPT) modules
- Specific/collections of user stories
- Assigned work items
- Queries against the test asset database

Using high-level business-level keywords, such as `Login.Process`, allows complexity hiding and reusable encapsulation. `Login.Process` contains a number of low-level keywords, such as `Enter Username Text` and `Press Login Button`.

The collection of application keywords represents natural domain-specific languages that translate into a number of lower-level actions performed before and after the core event. This included checking that the pretest and posttest conditions were met and the actions and reactions, including any popup/error recovery, were processed correctly.

Good Point

The more automation code is reused, the more worthwhile it is to build in recovery from unexpected errors or events and the more robust the scripts are.

Using this approach meant we had a foundation upon which to design and develop reliable, domain-specific, and reusable test scenarios before the release of the SUT.

Writing the test scenarios (manual or automated) as business-level keywords combined with natural language made it accessible to application domain experts, business analysts, and developers. The test language was self-validating and human readable, which removed the requirements to educate the end user in the tool. The verbs and nouns in the domain-specific language were written in natural language using context-sensitive validation. This improved the utilization of resources by encouraging joint collaboration between multifunctional teams while supporting behavior-driven development (BDD).

The Scrum team was made up of a number of multidiscipline team members (business analysts, testers, and developers) sharing the various roles of the test design phase without any previous automation experience. This allowed teams to collaborate and share test assets such as BPM/BPT modules, user stories, and work items. They could also run queries against previously created test scenarios and reuse shared test cases and steps.

The flexibility of having the test scenarios stored in a database also allowed for full/partial fallback support for legacy manual testing because the test data could be easily extracted into a traditional test scenario format. It was easy to read because of the use of natural language combined with valid test data that could be easily used in manual test execution.

Good Point

Automated tests should be accessible and readable by all and should enable the tests to be run manually if necessary.

In summary, this approach of managing centralized test assets to generate sanitized test scenarios validated against business rules provided ready-to-use tests and data in the correct format. This was evidenced by the ability to generate tests featuring over 10,000 ticket number combination states covering all possible combinations of ticket types and amounts (this excluded specialized test runs such as boundary and negative tests, which were run separately) before the SUT had even been written.

29.3.4 Problems and Solutions

We found problems stemming from procedures not being followed consistently. For example, changes to the functionality of reusable test components' jigsaw pieces were not being checked in correctly. This was caused by not having an enforceable gated check-in procedure and consequently resulted in limited reusability of some of the test components. The problem was solved by enforcing the check-in procedures in the Configuration Management tool.

Lesson

Automation development requires the same discipline as software development.

It became apparent when the testware framework entered the execution phase and was distributed across a pool of remote test clients generated by a virtual machine (VM) dispenser that there was limited direct visibility into the test execution status.

While it was relatively easy to identify primary core modules failing on startup, more subtle changes to reusable test components were much harder to spot. The requirement for a test asset loader to validate the current SUT build against the test asset database before execution could have prevented this.

Without the ability to monitor runtime failure, especially controlled failure (i.e., scenario recovery), a significant amount of execution time was wasted. For example, a discrete change to a test component could cause a false-positive error, which in turn caused the testware framework to repeatedly retry the current test scenario before attempting to continue through the remaining test cases. What was needed here was a suitable real-time dashboard that could provide notifications regarding the SUT health as well as progress of test client execution.

We solved this problem by devising a way to flag the overall status of a test set—In Progress, Ready, Repair, or Blocked—to keep the tester informed. This would affect the current test run and associated test client VM's state where, for example, a Blocked status did not allow the test run to be continued until the necessary pretest conditions were met (e.g., the Lotto product draw had been successfully processed).

Tip

Keep track of things that affect the test execution to avoid wasting time running tests that will fail for reasons you are not interested in. There is a danger, however, that tests that are turned off will never be turned back on.

29.3.5 Results of Our First Day Automation Approach

This approach worked extremely well, and we realized a good return on investment (ROI) for the additional effort in developing the framework.

Once the release was delivered, the execution was run constantly, day and night. This was made possible by having dedicated resources available during the day to deal with basic debugging of failed scripts and execution. Developers based in another time zone were also available in the evening to maintain the framework and provide additional support for improved test coverage.

Overall, this approach was found to work well in this case study by demonstrating its innate advantages, reflected in what I like to call the approach: Hybrid Keyword Data-Driven Automation Framework.

- *Hybrid:* Utilizing the best technologies and resources to do the job.
- *Keyword:* Creating simple and robust test scenarios written in business-level keywords combined with natural language.
- *Data:* Effective use of dynamic business data to provide an input source.
- *Driven:* Reusable component modules and libraries to provide reliable processing of generic actions, objects, and events.
- *Automation:* That is collaborative, distributed, and scalable.
- *Framework:* Independent of application, technology, or environment under test.

The best aspects of these proven approaches demonstrate how they have evolved over the past decade; this echoes some of the progress toward leaner and more agile business methodologies. They are in a constant state of evolution—just as the underpinning technologies evolve over time.

A significant benefit was that the framework had the ability to support multiple browsers, platforms, and technology stacks under a unified engine with the capability to deal with generic object classes as well as application-specific classes.

29.4 Attempting to Get Automation Started

Tessa Benzie, New Zealand
Test engineer

My company was keen to get into test automation. We had a couple of days of consultancy (with one of the authors of this book) to explore what we wanted and the best ways of achieving our goal. We discussed good objectives for automation and created an outline plan. We realized the need for a champion who would be the internal enthusiast for the automation. The champion needed to be someone with development skills.

We hired someone who was thought to be a suitable candidate to be the champion for our initiative. However, before he could get started on the automation, he needed to do some training in databases, as this was also a required area of expertise, and we needed him to be up to speed with the latest developments there.

After the training, he was asked to help some people sort out some problems with their database, again "before you start on the automation." After these people, there were some other groups needing help with databases. When do you think he finally started automation? As you may have guessed already, he never did! Of course, it can be very hard for a new person to say no, but the consequences should be pointed out.

A few months after that, a new test manager came in who was keen to get the automation started at the company. He did some great work in pushing for automation, and we chose and acquired a test tool that looked suitable. There were a couple of other contractors (the test manager was also a contractor) who were coming to grips with the initial use of the tool and began to see how it could be beneficial.

So we had a good start, we thought, to our automation initiative.

Shortly after this, there was a reorganization, and the contractors and test manager were let go. A new quality assurance manager was brought in, but test automation was not on her list of priorities. Some people were trying to use some of the automated tests, but there was no support for this activity. However, there were many, many tests that needed to be done urgently, including lots of regression tests. Now we had "football teams" of manual testers, including many contractors.

Lesson

Organizational support is needed to get a significant automation initiative started; be sure you choose a champion who will stay on task and be the driving force behind your initiative. Beware of knowledge walking out the door with your contractors—it's far better to involve permanent staff.

29.5 Struggling with (against) Management

Kai Sann, Austria
Engineer and test manager

I have had some "interesting" managers over the years who had some challenging effects on the way we did test automation.

29.5.1 The "It Must Be Good, I've Already Advertised It" Manager

We started software test automation in 2002. The management's intention was to reduce time for the system test. Furthermore, the management used automation as a marketing strategy before the automation was developed.

At this time, there was no calculation for return on investment (ROI). The approach was this: Software test automation must pay because manual testing is no longer needed. The goal was to automate 100 percent of all test cases.

I had a hard time explaining that software test automation is just one of many methods to achieve better software and that it is not free—or even cheap.

29.5.2 The "Testers Aren't Programmers" Manager

We started very classically and believed the promises of the vendors. They told us, "You only need to capture and replay," but we found this was not true. In our experience, this leads to shelfware, not success—it does not pay off.

After some trial and mostly error, we started to write automation code. At this point, we were far away from developing automated test cases. We needed some lessons in writing code. We were lucky to have very good mentors on our development team who taught us to write libraries and functions so we didn't have to code the same tasks repeatedly.

I had a discussion with my boss about what programming is. He explained to me that he consciously hadn't hired testers with an informatics degree because he didn't want to have more developers in his department. You can imagine his surprise when I told him that our automation code included libraries and functions.

He told his superiors that the testers do "advanced scripting" rather than coding because he was afraid that the testers would otherwise be forced to write production code!

Good Point

Beware of political issues and fears.

29.5.3 The "Automate Bugs" Manager

An idea provided by one manager was to automate bugs we received from our customer care center. We suffer the consequences to this day. How did this work? Our developers had to fix our customers' bugs. We were told to read this bug and automate this exact user action. This is where the consequences come in: Not knowing any better, we hardcoded the user data into our automation. After 2 years, we were one major release behind the development. We didn't know anything about data-driven tests at that time.

We were automating bugs for versions that were not in the field anymore. Most of these test cases still exist because we haven't had time to replace them.

Lesson

Holding the title of manager doesn't necessarily mean a person knows best (or anything) about test automation! You may need to educate your manager.

29.5.4 The "Impress the Customers (the Wrong Way)" Manager

My boss had the habit of installing the untested beta versions for presentations of the software in front of customers. He would install unstable versions and then call our developers from the airport at 5:30 a.m. and order immediate fixes to be sent to him by email.

Our programmers hated this so much that we introduced an automated smoke test. This test checks if we have a new build; then it installs the beta, and finally it checks the basic functions of our product. Our boss was told to only install smoke-tested beta versions.

Today we don't have this boss issue anymore, but we continue the automated smoke tests for our nightly builds because they provide us with a good guess about the state of our software. Here we really save money because smoke tests must be done anyway and we can provide our development with a lot of issues concerning the integration of new modules at an early stage. We expand this test every few months. The coolest thing is that we are informed about the latest test results by email.

So in spite of some challenging managers, we are now doing well with our automation!

Good Point

Sometimes an approach adopted for one reason turns out to be good for other reasons.

29.6 Exploratory Test Automation: Database Record Locking

Douglas Hoffman, United States
Consultant and speaker

Pre–World Wide Web (late 1980s), I was the quality assurance (QA) manager, test lead, and test architect for the second-largest relational database management system (RDBMS) software company at the time. Before landing the job, I got my bachelor's degree in computer science, started out as a systems engineer, and had worked my way up into hands-on engineering services management (QA, tech support, sales support, and tech writing).

The company had about 300 employees, mostly in San Mateo, California. The relational database engine had evolved over more than 10 years and had a well-established, fairly stable codebase. At the time, the code had to be ported across 180 hardware/software platforms (most were variations of UNIX). The QA team was small (initially with a ratio of about 1:20 with developers, growing to ~1:5 over 18 months) and nearly completely focused on testing. Most new testers were recruited from other companies' development organizations.

To support the large number of versions, the product was developed using internal switches to enable or disable different code functions. Therefore, most of the porting was done by selecting the proper combinations of switches. This meant that once features or bug fixes had been incorporated into the base code, the various ports would pour into QA.

Because the test team was small, we spent almost all our time running tests on the various platforms. Little time was available for design and implementation of tests for new features. A thorough test run for a new release on a platform could take 2 weeks, and the dozen testers could receive 150 versions in a few weeks. The management and technical challenges dealing with this situation are other case studies in themselves. This case study focuses on one particular exploratory automated test we created.

Unlike regression tests that do the same thing every time they are run, exploratory automated tests are automated tests that don't have predetermined results built in; they create new conditions and inputs each time they're run. Exploratory test automation is capable of finding bugs that were never specifically conceived of in advance. These are typically long-sequence tests and involve huge numbers of iterations because they are only limited by available computer resources.

Tip

When the input space of an application is huge, exploratory automated tests can find defects that are difficult to find manually.

29.6.1 The Case Study

Bug being sought: Errors in database record locking (failure to lock or release a lock on a record, table, or database).

When I took over the QA organization, the existing test cases were simple applications written in the various frontend languages, mostly our proprietary SQL. Most existing tests were applications and data sets collected from customers or scripts that verified bug fixes. Automation was achieved using a simple shell script that walked through the directories containing the tests and sequentially launched them. Most of the testers' efforts were in analyzing the results and tweaking the tests to conform with nuances on the various platforms.

One area of concern that wasn't well tested using our regression tests was record locking. A few intermittent bugs that locked up the programs or clobbered data had been found in the field. The locking mechanism was complex because of the distributed nature of the database. For example:

- Parts of the database might be replicated and the "master" copy moved around as requests were made.
- Different parts of the database needed to communicate about actions before they could be completed.

- Requests could occur simultaneously.
- Common data could be updated by multiple users.
- One user's request for exclusive access (e.g., to update part of a record) could cause some other users' requests to wait.
- User requests for nonexclusive access might proceed under some circumstances.
- Non-overlapping parts of interlocking records might be updated by different users.
- Timeouts could occur, causing locks and/or requests to be cancelled.

Most multiuser database systems at the time were hardwired and LAN based, so Internet and cloud issues weren't part of the picture. (This was before widespread Internet use, browsers, or the World Wide Web.) Frontend programs were built out of compiled or interpreted programs written in proprietary languages. The use of LANs meant that interrupt events came directly through hardware drivers and were not processed by higher-level system services.

Prior to the test automation project, the straightforward locking sequences were tested using manual scripts on two terminals. For example,

1. One user would open a record for nonexclusive update (which should lock the record from being modified but allow reading of the record by other users).
2. A second user would open and attempt to modify the record (thus successfully opening but then encountering a record lock).
3. Another test would have the first user opening for nonexclusive read (which should not lock the record and should allow reading and modifying of the record by other users).
4. The second user would read and update the record (which should work).

The regression tests confirmed the basic functioning of the lock/unlock mechanisms. Only a subset of condition pairs could be manually tested this way because of the amount of time it took, and complex sequences of interactions were out of the question. Figure 29.2 shows an example of the interaction of two users attempting to access the same record.

In relational databases, updating a record can write data in many tables and cause updates to multiple indexes. Different users running different programs may reference some common data fields along with unique data. The data records are contained in multiple files (called tables), and programs reference some subset of the fields of data across the database. Intermittent, seemingly unreproducible problems could occur when the requests overlapped at precisely the wrong times. For example,

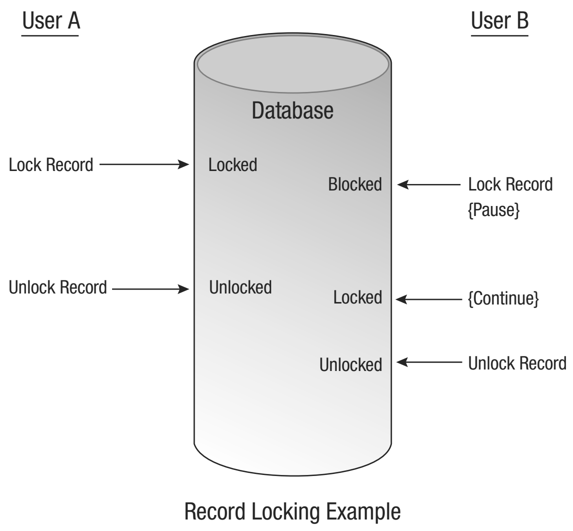

FIGURE 29.2 Record locking example

the second read request might come in while the first was in the process of updating the database record. There might be tiny windows of time during which a lock might be missed or a partially updated record returned. These kinds of combinations are extremely difficult to encounter and nearly impossible to reproduce manually. We decided that the best way to look for errors was to generate lots of database activities from multiple users at the same time.

Good Point

Good candidates for automation are tests that are difficult to run manually and those that are too complex for manual testing.

The challenge was to create multithreaded tests that could find timing-related problems of this type. The goal was to produce tests that would generate lots of conditions and could detect the bugs and provide enough failure information to the developers so they could isolate the cause and have some chance at fixing the bugs.

Automated tests: We created a single program that accessed a database using various randomly selected access methods and locking options. The test verified and logged each action. We then ran multiple instances of the program at the same time (each being its own thread or on its own machine). This generated huge numbers of combinations and sequences of database activities. The logs provided enough information to recreate the (apparent) sequence of database actions from all the threads. If no problems were detected, we discarded the logs because they could get quite large. While the tests were running, a monitoring program observed the database and log files to ensure that none of the threads reported an error or became hung.

Oracle: Watching for error returns and observing whether any of the threads terminated or took excessively long. The test program did very trivial verification of its own activities. By the nature of multiple users updating overlapping data all the time, data changes might be made by any user at any time. We couldn't reliably confirm what was expected because some other user activity might change some data by the time the test program reread it. Because most database activities completed in fractions of seconds, if there was no lockout, the monitor program checked for multisecond delays on nonlocked transactions or after locks had been logged as released.

Good Point

A test oracle, especially for automated exploratory testing, may just be looking for system failures rather than functional correctness. Use an oracle that is appropriate to the testing you are doing.

Method summary:

1. Created a program that independently, randomly accessed records for update, read, insert, and delete. Different types of record locking were randomly included with the requests. Each program logged its activity for diagnostic purposes and checked for reasonable responses to its requests.
2. Ran multiple instances of the program at the same time so they potentially accessed the same data and could interfere with one another. Access should have been denied if records were locked by other processes and allowed if none of the other threads was locking the referenced record.
3. Created and ran another program to monitor the log file and the database engine to detect problem indications and shut down if a problem occurred.

Because the threads were running doing random activities, different combinations and sequences of activities were generated at a high rate of speed. The programs might have detected errors, or the threads might hang or abnormally terminate, indicating the presence of bugs. Each instance of the test generated large numbers of combinations of events and different timing sequences. The number of actions was limited by the amount of time we allocated for the lock testing. We sometimes ran the test for a few minutes, but at other times it could run an hour or longer. Each thread might only do hundreds of database actions per second because of the time it took for waiting, error checking, and logging. We ran from three to a dozen threads using multiple networked systems, so a minute of testing might generate 100,000 to 300,000 possible locking conditions.

Results: We caught and were able to fix a number of interesting combinations, timing-related bugs, and one design problem. For example, a bug might come up when:

- User A opens a record for update.
- User B waits for the record to be unlocked to do its own update.
- User C waits for the record to be unlocked to do its own update.
- User A modifies the data but releases the lock without committing the change.
- User B modifies the data but releases the lock without committing the change.
- User C updates the data, commits it, and releases the lock.
- User C's data was not written into the database.

I was surprised because a few of the timing- and sequence-related bugs were not related to the record locking itself. If the commits occurred at the same time (within a tiny time window), the database could become corrupted by simultaneous writes of different records by multiple users in a single table. Although the records were not locked because the users were referencing different rows, the data could become switched.

We couldn't be certain that we caught all the bugs because of the nature of these kinds of timing- and sequence-related problems. Although millions of combinations were tried and checked, there were myriad possibilities for errors that we couldn't detect or didn't check for. Trillions of combinations wouldn't amount to a measurable percentage of the total number of possibilities. But, to the best of my knowledge, there were no reported field problems of that nature for at least a year after we created these tests. (I moved on and no longer had access to such information.)

Good Point

The benefits of exploratory automated tests may be significant even if you can't know what you didn't test.

We didn't leave reproducibility to chance, although even running the same series of inputs doesn't always reproduce a problem. The at-random tests used pseudorandom numbers; that is, we recorded seed values to be able to restart the same random sequences. This approach substantially improves reproducibility. We generated a new seed first when we wanted to do a new random walk, and we reused the seed to rerun a test.

Tip

Even random tests should be reproducible in order to confirm bugs and bug fixes. Use the same starting point (the "seed") to reproduce the same random test.

The archiving system is critical for tracing back to find the likely cause and also as the test oracle. Much of the data being recorded can be checked for consistency and obvious outliers to detect when likely bugs are encountered. I tend to log the things developers tell me might be important to them as the test progresses and then "dump the world" when a bug is suspected.

We used some functional test suites that were available for SQL, including from the National Bureau of Standards (which later became the National Institute of Standards and Technology), but they only checked basic functionality. We used them to some degree, but they were not random, asynchronous, or multithreaded. TPC-B wasn't created until several years later.

Recognizing the root cause of reported bugs required serious investigation because the failures we were seeking generally required multiple simultaneous (or a specific sequence of) events. Many of the factors we looked at were environmental. We were primarily looking for fundamental violations of the locking rules (deadlocks and data corruption), so recognizing those failures was straightforward. Identifying the cause was more difficult and frustrating. It sometimes took a lot of investigation, and once in a while, we gave up looking for the cause. This was frustrating because we knew there was a bug, but we couldn't do anything about it other than look for other symptoms. Most of the time, though, the developers were able to identify the probable cause by looking for the possible ways the failure could have happened.

Good Point

Unreproducible failures are worth reporting because sometimes developers can trace the cause in the code if they know that something is wrong.

29.7 Lessons Learned from Test Automation in an Embedded Hardware–Software Computer Environment

Jon Hagar, United States
Engineer, trainer, and consultant

Embedded systems comprise specialized hardware, software, and operations. They come with all of the problems of normal software, but they also include some unique aspects:

- Specialized hardware that the software "controls" with long and concurrent development cycles.
- Hardware problems that are "fixed" in the software late in the project.
- Limited user interfaces and minimal human intervention.
- Small amounts of dense, complex functions often in the control theory domain (e.g., calculating trajectories, flight dynamics, vehicle body characteristics, and orbital targets).
- (A big one) Very tight real-time performance issues (often in millisecond or microsecond ranges).

Products that make up embedded software systems now span the automotive, control, avionics, medical, telecom, electronics, and almost every other product domain one can think of. I have been involved in space avionics (guidance, navigation, and control software), but many of the approaches and lessons learned are applicable to other embedded software systems. In this section, we use examples drawn from a hypothetical but historically based space flight software embedded system.

The goal of verification, validation, and testing (VV&T) is to show that embedded software is ready for use and the risk of failure due to software can be considered acceptable by the stakeholders.

Development programs can be small—for example, 30,000 source lines of code (with staffs of 10 to 60 people)—yet these programs are time and computationally complex and are critical to the successful control of the hardware system.

29.7.1 VV&T Process and Tools

We typically have four levels of testing and tools that support each level. The lowest level is probably the most different for embedded systems because it is nearest to the hardware. It uses a host/target configuration and cross-compiled code (including automation code). Cross-compiling is where source code is compiled on one computer, not into the binary (executable) of that (host) computer but rather into binary executable code that will run on a different computer (the "target") that is too limited to be able to run a compiler on. Our testing at this level aims to check against standards and code coverage as well as requirements and design and is automated by the developer.

We call this "implementation structural verification testing" (some places call this unit testing). This testing is conducted with a digital simulation of the computer and/or a single-board target hardware-based computer.

The implementation test tools were customized in the beginning, but other off-the-shelf tools were added later. Examples include LDRA TBrun, Cantata, and AdaTEST. The project used both test-driven development and code-then-test implementation approaches. The comparison and review of results, which include very complex calculations, is done using test oracle information generated from commercial tools such as MATLAB, BridgePoint, and Mathmatica.

The middle level, which we call design-based simulation tools, uses tools that are based on software architecture structures and design information, which have been integrated across module boundaries. These tools allow the assessment of software for particular aspects individually. In some projects, model-based development tools, BridgePoint, and MATLAB were used, and this enabled the integration efforts to go better than in past systems, because the models enforced rules and checks that prevented many integration defects from happening in the first place.

Tip

Using models can help to prevent and eliminate defects that otherwise would be found in testing (or not found), but nothing guarantees that you find 100 percent of them.

The next level is requirements-based simulation (scientific simulation tools). These simulations (driven by models) are done in both a holistic way and based on individual functions. For example, a simulation may model the entire boost profile of a system with full vehicle dynamics simulation, and another simulation may model the specifics of how the attitude thrust vector control works.

This allows system evaluation from a microscopic level to a macroscopic level. The results from one level can be used as automated oracles to other levels of VV&T test supporting "compare" activities.

This approach of using simulation/models to drive and analyze test results comes with a risk. There is the chance that an error can be contained in the model or tool that replicates and "offsets" an error in the actual product (a self-fulfilling model result). This is a classic problem with model-based test oracles. To help with this risk, the project used the levels of testing (multiple compares), a variety of tools, different VV&T techniques, and expert skilled human reviewers who were aware of this risk. These methods, when used in combination with testing, were found to detect errors if they exist (one major objective) and resulted in software that worked.

Finally, at a system level, VV&T of the software uses actual hardware in the loop and operations. An extensive, real-time, continuous digital simulation modeling and feedback system of computers is used to test the software in a realistic environment with the same interfaces, inputs, and outputs as in the actual system. The system under test runs in actual real time; thus there is no speed-up or slow-down of time due to the test harness. Additionally, with hardware in the loop and realistic simulations, complete use scenarios involving the hardware and software could be played out with both for typical usage scenarios (daily use) and unusual situations such as high load, boundary cases, and invalid inputs.

29.7.2 Lessons Learned

This section summarizes some general observations that the projects had during the initial setup and use of automated VV&T tools:

- *Training:* It is important to allow both time and money for training on tools and testing.
- *Planning:* Tools must be planned for and developed like any software effort. Automated VV&T tools are not "plug and play." To be successful, plan for development, establish a schedule and budget, integrate with existing processes, plan the test environment, and also test the test tools. Test tools must be "engineered" like any development effort.
- *Have an advocate:* Test tools need a champion in order for them to become incorporated into common use. The champion is someone who knows the tools and advocates their use. Success comes from getting staff to think "outside the automated tool box." The new tools must "integrate" with the existing staff, which means education, mentoring, and some customization. Advocates work these issues.
- *Usability of a tool must be reasonable for the users:* While people will need training on tools, and tools by nature have complexities, a tool that is too hard to use or is constantly in revision by vendors leads to frustration by users that, in the extreme, will lead to shelfware. Ensure that the user interface is part of the selection evaluation before purchasing any tool.

Good Point

Usability of the tools is important—even for "techies."

- *Expect some failures and learn from them:* Our project explored several tools that were abandoned after an initial period of time. While failure is not good, it is really only total failure when one does not learn from the mistake. Also, management must avoid blaming engineers for the failure of an idea because doing so stifles future ideas.

Good Point

If you learn from your mistakes, you have not totally failed. Any failure or mistake becomes a source of information to share.

- *Know your process:* Automated test tools must fit within your process. If you lack process, just having tools will probably result in failure. Expect some changes in your process when you get a new tool, but a tool that is outside of your process will likely become shelfware.
- *Embedded systems have special problems in test automation:* Despite progress, automated test tools do not totally solve all embedded VV&T problems. For example, our projects found issues in dealing with cross-compiling, timing, initialization, data value representation, and requirements engineering. These can be overcome, but that means vendors have more functions to add and projects will take more money and time. Plan for the unexpected.
- *Tools evolve:* Plan on test tool integration cycles with increments.
- *Configuration management (CM):* Even with VV&T tools, projects need to manage and control all aspects of the configuration, including the test tools as well as the test data.

29.7.3 Summary of Results

Although I am not permitted to reveal specific data, when compared to custom-developed tools and manual testing, establishing an automated commercial-based VV&T environment took about 50 percent fewer people. The projects tend to take these savings to create more and/or better automated tests. While adding to test automation, the projects maintained and improved functionality and quality. Further, maintenance-regression costs decreased because vendors provided upgrades for a low annual fee (relative to staff costs for purely customized tools). Commercial tools have a disadvantage of lacking total project process customization, but this has proven to be a minor issue as long as the major aspects of the process were supported by the tools.

Additionally, the projects reduced test staff work hours by between 40 and 75 percent (based on past VV&T cycles). We found that our test designers were more productive. We created the same number of tests and executed them in less time and found more defects earlier and faster. We had fewer "break-it, fix-it" cycles of regression testing, which meant that less effort was needed to achieve the same level of quality in the testing and the same defect detection rates.

In an embedded software VV&T environment, automated test tools can be good if you consider them as tools and not "magic bullets." People make tools work, and people do the hard parts of VV&T engineering that tools cannot do. Tools can automate the parts humans do not like or are not good at. Embedded projects continue to evolve VV&T automation. VV&T automation tools take effort, increments, and iterations. Tools aid people—but are not a replacement for them.

Good Point

The best use of tools is to support people.

29.8 The Contagious Clock

Jeffrey S. Miller, United States
Developer

Sometimes a good testing idea is contagious. Once it meets one need in your system, other uses may emerge that were quite unexpected when you began.

29.8.1 The Original Clock

I had just been hired at Google as a developer on a project during its preparation for public release. The system under development embedded a timestamp when recording certain events. Depending on how long it had been since the events were recorded, the system needed to present, interpret, or process the events in different ways.

The project had a strong mandate for developers to demonstrate the features they created via automated unit and system tests. As my first development task, I took on the job of designing and coding an application clock to make developer testing of time-based behavior simpler. In production, the application clock follows the system clock, but for testing, it wraps a test clock that can be manipulated to simulate the passing of minutes, hours, or days.

29.8.2 Increasing Usefulness

At first, the application clock was used for automated testing of portions of code encapsulating the core logic for time-dependent features. However, the system under development could be driven as a whole via a scripting language that could simulate one or more users interacting with the system to accomplish a set of tasks. Script-driven system tests were the common property of developers, feature owners, and a team of testing specialists. The testing team used script-driven system tests alongside manual verification to exercise the system in detail before each version was released to production. Soon I helped add commands to the scripting language to control the clock, allowing nonprogrammers to set up scenarios that included the passage of time.

29.8.3 Compelling Push

The original application clock was limited by design so that the clock could never be manipulated in the production system and thereby create troublesome inconsistencies. However, the testing team needed to exercise the system and its features interactively in a staging environment similar to the production setup. However, for testing time-based behavior, sometimes they set up a scenario before a weekend and returned after the weekend to verify the correct behavior. Other times the testing team changed the system clock so that the application would pick up the changed time and demonstrate the desired behavior. Both of these techniques were laborious and error prone, with the system clock manipulation frequently causing side effects that would ruin the test.

At the request of a primary tester and another developer familiar with the application clock, I revisited the application clock design. By this time, the system supported a mechanism for enabling and disabling features in production without having to redeploy a new system binary. This mechanism allowed me to guard the application clock from being manipulated on the actual production servers while allowing the testing team to control time interactively on their own simulated production servers.

29.8.4 Lessons Learned

The main thread of this story follows a version of a software developer's adage: "Wrap external dependencies." While the runtime library is normally considered internal, the clock it provides is a service outside the system. When the passage of time is important in system logic, wrapping the clock is a beneficial move.

The unexpected bonus was that adapting to successively larger scopes (isolated code, scripted captive system, interactive deployed system) provided benefit to more and different groups of people and for different types of tests. Although the larger scopes required modestly more architectural plumbing, in each case the wrapped clock fit into configuration systems that had been built to bring other benefits to the system. With hindsight, it would have been better to build them earlier had we known more of the operational and testing uses for the application clock.

I've now moved on to other work within the company, but I can see the application clock has been maintained and adapted to fit the system's new configuration mechanisms. I'm glad it continues to prove useful.

Lesson

Look for wider application for any useful utilities that help to automate some aspect of testing.

29.9 Flexibility of the Automation System

Mike Bartley, United Kingdom
Lecturer and consultant

We developed our test automation system ourselves and devised ways to adapt our automated testing to be more efficient in ways that we have not seen elsewhere.

Because we had already developed an inhouse software version control and build system, it was fairly easy to integrate our automation tool with our build system. This made our testing more efficient because we could selectively test only those modules that had changed, as shown by our source code dependency tree. If nothing that a particular test depended on had changed, that test would not be executed. This dramatically reduced the build and cycle time and thus allowed us to put in place continuous integration of builds and tests. We did keep an option that forced all tests to rebuild and run if we wanted to run a full regression test.

We made it easy to remove tests from the test suite when a test needed updating because of changes in the software it was testing. Although we were then cutting down on the test coverage because the test(s) were not run, it meant that the maintenance of those tests didn't have to be done immediately, thereby stopping the rest of the test suite from running.

We extended this to a way of "banning" specific tests for various reasons:

- The software has changed, but the tests have not yet been updated.
- A test is known to be giving a false failure (i.e., it fails but it should pass).
- A test is not restoring test machines to a known good state.

This idea proved to be a major benefit to our efficiency.

Tip

Being able to selectively choose tests to run or not to run can make the automation quicker and more efficient. Make it clear which tests are active and which are not.

29.10 A Tale of Too Many Tools (and Not Enough Cross-Department Support)

Adrian Smith, United Kingdom
QA lead

I have been involved in five automation projects over 5 years, with varying degrees of success.

29.10.1 Project 1: Simulation Using a DSTL

The first project was written in Python and batch scripts, running functional and performance tests on 85 to 145 PCs, simulating more than 20,000 machines. It was not originally an automation project, but I made it so. What started as a simple control program ended up growing into a fully flexible domain-specific test language (DSTL), as it would now be called, that enabled the tester to write test steps in a simple though unstructured keyword/parameter language. Expand the tests, alter them, chain them, and add new elements to the language as needed. The potential 9-month project was still being used 6 years later, It has ended up having a much better ROI than expected as its scope has increased over time. Thousands of man hours were saved and vastly more test runs were performed than a could have been run manually and with fewer execution errors.

About halfway through the automation project, my manager wanted me to do some manual testing for a different project because it was way behind. I knew this would be the end of the automation, so I managed to convince him that it would be better for me to stay with this (ad hoc) project—and this contributed to its success.

Lesson

Have the courage to argue your position with your manager.

29.10.2 Project 2: Testing a GUI Using TestComplete

The second automation project was to automate system-level applications. A tool was bought to experiment with: TestComplete 3. I had high hopes for another success using a DSTL for the testers, but this would take a long lead time to build. We then came across problems of automating GUI interfaces written in an automation-unfriendly way. I naively asked development to assist by modifying their code to help with the automation but was flatly refused. I had no support from management for this, so I had to go it alone. I probably should have stopped there and then.

Lesson

Without the cooperation of developers, test automation will be more difficult than it needs to be.

But I didn't. I persevered with the DSTL framework, though with little abstraction because I wanted to have something working sooner for management. At the time the first beta was just about ready, a new director of testing was appointed. The good news was that he thought automation was a good thing. The bad news was that he decided we needed to get "the right tool" with a single solution of manual test creation, results gathering, and reporting. I had to suspend my work with TestComplete and was given a 2-month task to evaluate a number of GUI automation tools. The final three were Rational Robot, Quality Centre QTP, and guess what: TestComplete. After the evaluation, I thought TestComplete was the most flexible and wanted to continue with it. The company thought differently, so this framework was never completed.

29.10.3 Project 3: Rational Robot

A 3-month proof-of-concept was then initiated for Rational Robot. If I had got further in the previous project, I could have at least reused the tests written. It was decided to do something similar with this tool, framework, abstraction, and tests being a thin layer on top. After 8 months, another automator and I had abstracted the tests and had a GUI object action library that could be generated automatically. Many hundreds of lines of code were automatically generated to do simple GUI actions as click a button or check a textbox. All that was changing was which control in which window to use. We had a good feeling about this framework because it was simple, and we were just starting to settle on a project to automate when, at this point, management decided to do a proof-of-concept for QuickTest Professional (QTP).

29.10.4 Project 4: Final Python Project and QTP Proof-of-Concept

Management were now getting involved and wanted to see some ROI that could be quantified to justify the purchase of this tool. We set to work on a single-use framework in Python, eventually automating 15 end-to-end smoke tests using GUI libraries. I had made a Python frontend so that testers could create and maintain tests without needing a lot of technical knowledge. The number of bugs that it found was too low to justify extending this automation to other areas. However, we were beginning to get occasional cooperation from developers. There were a couple of interfaces that could be called directly from Python to C written specifically for us to enable the tests to function.

We had one problem that we lost many days trying to figure out: The tool would crash but didn't report any error. It turned out to be a bug in the tool itself.

Good Point

Don't forget that the tool itself is also software and may have bugs.

For the proof-of-concept project for QTP, we had trouble trying to work with QTP in the way we wanted to, and a lot of time was wasted coming to grips with it. Eventually, we found a workaround to allow us to put many methods in one QTP file. At the end of this proof-of-concept, I would still have opted for one of the *other* tools.

Management chose QTP, and we had a real project to do with deadlines and end dates, so many of our ideals were curtailed, sidelined, or dropped. We again ran into problems with GUI objects and no help from developers.

29.10.5 Project 5: QTP2

With a new release of QTP, we tried to integrate our framework and Python code so that test results would be received centrally while still allowing us to launch tests including rebooting machines. This was using VMware virtualization and CruiseControl. We added extensively to application libraries, which were QTP libraries that did lots of specific tasks in a GUI, passing in a number of parameters. We also wanted to bring the test creation tool up to date so that the testers could use the automation easily. The thought behind this was that the easier it was to write a test, the quicker it would be to add tests, while the application libraries could be maintained by the automators.

However, management didn't want "extraneous time" spent on this perceived nonproductive activity!

Lesson

Managers sometimes think they know best even when they don't. As an automation champion, dare to say "Mission Impossible" when necessary.

The way we automators wanted it was that testers would not have to learn much programming, but because there would be no tool to help with creating tests, then testers would have to learn programming and know much more about the internal workings of the automation system. This is not a bad thing, but with a lot of pressure on the test department, it seemed (and was proven to be true) that testers rarely had time to dedicate to automation programming before being taken off again by another project. In a lot of cases, they never made it to do any automation because of deadline changes. At this time, automation still was not at the forefront of management thinking, but it was definitely starting to get their notice.

Progress was being made, libraries were expanding, and the tests were nearing completion, having overcome many of the problems of GUI fragility.

Now a problem that we automators had "forgotten about" came back. For the first time in a couple of years, the GUI interfaces began to be amended or overhauled on a much more frequent but ad hoc basis (to us automators). We were not informed of changes as they happened, so our tests started failing, and it took a long time to find out why. After 2½ months of battling this upset (hundreds of changes to GUIs rendered even the smoke tests useless), I called a halt.

Lesson

Changes to the user interface can derail the automation if it cannot cope with such changes.

29.10.6 The End of the Story

Of the five automation projects I was involved in, only the first one achieved success. It was non-GUI and an isolated project. The others failed because of what seemed to be management decisions and lack of cross-department cooperation, but perhaps better communication would have helped.

Management had a bit of a shock and a rethink about automation after all the previous high-profile problems. Automation is now a deliverable for *developers*—one of the key problems before was that there was no incentive for developers or development managers to support or even cooperate with automators, as they had their own targets. Direct GUI automation has been abandoned, and automation is now at API level.

The final irony is that developers now have to maintain the APIs and automation code; if only they had agreed to maintain object libraries or had added a few lines to object maps earlier, there would have been less work for them now.

29.11 A Success with a Surprising End

George Wilkinson, United Kingdom
Test manager, trainer, and consultant

This anecdote describes some of my experiences on a large test automation project undertaken in 2007 and 2008. This project was to automate the core processes of the system validation tests of the National Health Service (NHS) Care Records System (CRS) application as rolled out within England by a large health IT Systems Integration company. This was being undertaken as part of the wider National Programme for IT (NPfit). The study covers 8 months of continuous progress, though with a surprising end.

An automation team was formed from a number of locations, including the North of England, the Midlands, and the West Country. Rather than looking for an exact skills match, we wanted people experienced in the CRS application who were enthusiastic about getting involved in automation. Because the team was geographically distributed, we decided to meet most weeks in a geographically central location for 2 days.

Good Point

Team-building is important for automation teams, too, especially when they are geographically dispersed.

29.11.1 Our Chosen Tool

TestStream was a commercial validation suite from Vedant Health, a United States company specializing in test health-care automation targeted at laboratory information systems (LIS) and health informatics systems (HIS). Our representative from Vedant traveled from the United States to start the project going and to run the training in the product set and the TestStream methodology.

Good Point

Take advantage of learning from others when the opportunity arises.

One of the useful features of TestStream was called Scenario Builder. It provided a way to construct automated patient journeys, which are made up of a number of predefined actions. The analyst simply pulls together these actions to create a longer test. There are over 600 actions for our CRS application system, and they include elements such as Register a Patient, Add Allergy, Schedule Surgery, and Check in a Patient. The Scenario Builder allows the sequence of events to be defined and viewed as a true patient journey.

No scripts, scripting, or further script development was required by either my team or Vedant Health, because the Scenario Builder's actions provided the components or scenarios required. The only requirements were a solid familiarity with the application under test and a thorough understanding of the test case (normally a patient journey).

We built a comprehensive library of automated scripts and devised standards and procedures about how they were to be stored and maintained. We developed a customized comparison and data collection tool, which we called CAT (collection analysis tool).

29.11.2 The Tool Infrastructure and an Interesting Issue as a Result

The product was installed and accessible by the user via a secured network to servers running virtual machines (VMs), as shown in Figure 29.3. Access to the VMs and thus to the test environments was provided to both the automation team running tests and company IT support staff.

Vedant's access for support could be from anywhere in the world because some of the highly experienced Vedant support staff moved around the world assisting other clients. This required remote access to our infrastructure, but we soon discovered that it didn't work. The system was so secure (in order to prevent fraudulent access into any test environment that may hold live patient data) that it prevented the remote access facility from working.

Good Point

Don't forget to test your support facility, especially if you have stringent security requirements.

We resolved the issue by allowing both companies independent access to another test system that was clean of any patient data. This solution was foolproof from a security perspective but provided only limited support, which was to be mitigated by the test system holding the same application version that the majority of systems were holding in the field. Although the solution was not perfect, because the deployments were not always running the same system version, it was a step in the right direction—and one on which we could make progress.

Looking back, we realized that no feasibility study had been conducted on support, which could have prevented the remote access issue from arising.

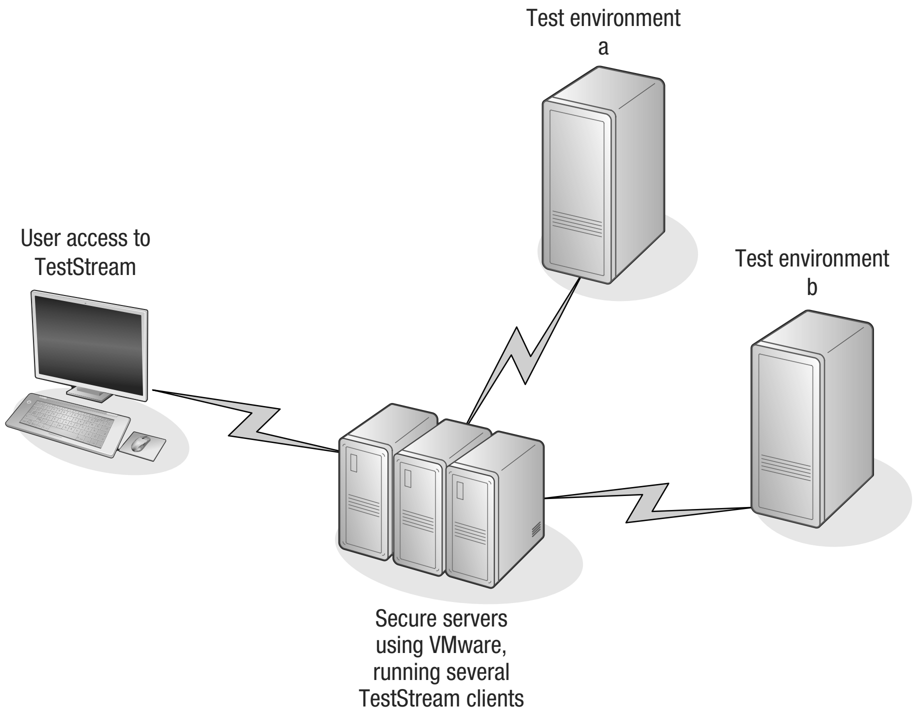
Test environment a
User access to TestStream
Test environment b
Secure servers using VMware, running several TestStream clients

FIGURE 29.3 TestStream infrastructure

29.11.3 Going toward Rollout

Over the next 3 to 4 months, the team grew from 6 to 10, with an additional four part-time support members. We produced a catalog of the automation tests that were available to the deployment projects to build their own scenarios. As we progressed with the pilot, we identified data and configuration requirements that were localized to the individual projects as they moved away from a standard. This meant that our current generic approach needed to be tailored for the deployment-specific test environment. What we had done was created a process but lost some sight of our individual customer's requirements.

Lesson

Don't invest too much effort in designing your automation without testing it out in a real situation.

We ran a sample of the data collection and clinical ordering features of the CRS for a particular deployment. This was a great success because we found many defects that were thereby prevented from entering the live environment. We found between 10 and 100 defects on well-built and established test environments and thousands on other environments.

We published a report to the stakeholders showing how we added value to the current manual test approach. We found that we could automate tests for around 70 percent of the installed CRS functionality and save approximately 30 percent of our current testing effort.

We now decided to initiate some public relations for the tool. We scheduled several educational sessions to explain the program and what we had been doing, to give stakeholders the opportunity to ask questions, and to gather feedback from the teams working on customer sites.

Lesson

You probably need to do more public relations and communication than you thought. Make pilots or demonstrations part of project milestones so that they are more visible.

I was quite surprised at how many people had a very different interpretation than we did of the product set and its intentions and of software test automation itself. Most people's experience with automation test tools is that they require constant scripting or maintenance to work. Fortunately, these sessions helped to convince people that our automation was an improvement on that.

We also dispelled some illusions and misperceptions about automation and set more realistic expectations. The public relations meeting also raised the team's confidence and gave them some well-deserved recognition.

The automation team were elated by the results from the pilot project and the fact we were now in the rollout stage. Their confidence was really growing; after all, they had made it work. TestStream was out there and making a real difference! We were positioning ourselves well, and the future, at last, after a good deal of effort, was looking more positive.

29.11.4 The Unexpected Happens

In late May 2008, after discussing our success so far and the rollout plans, the overall project was cancelled because of a breakdown in the contract with the systems integration company. Therefore, our automation project was also cancelled. I gathered my team together for the last team meeting and officially announced the cancellation. They had worked extremely hard, but the automation project was over; all those many late evenings, weekends, sheer determination, and extra miles traveling to make this work were now history. What a heartbreaking end to what should have been a great success.

Good Point

Sometimes in spite of a great effort, things don't go the way we expect because of factors entirely out of our control. Take what you can from the experience to use next time.

29.12 Cooperation Can Overcome Resource Limitations

Michael Albrecht, Sweden
Test manager, consultant, and speaker

I was in a test team that was testing technical aspects of banking system processing without access to a GUI. For this project, we needed not just domain knowledge but more technical skills than we had. Rather than take the traditional project approach and try to hire someone, we got everyone together, both testers and developers, and developed the skills we needed between us, although we did need to bring some testers with coding skills into the team.

We had no money for tools, so we just started to use the same open source tools that the developers were using. The difference in the way we used them was that we needed to do some coding to create scenarios for our tests rather than just exercising each one individually. We also needed to verify our expected results and double-check with the database directly.

We didn't spend any money on tools, but we did spend a lot of time (we also built our own performance-testing tool). Sometimes it is easier to explain (or hide) these costs: The purchase of a tool would appear to be a large single cost, but time being spent over months or years doesn't appear to be as significant even if it is the same or even more money!

We found that close cooperation with the customer and working as a team enabled us to succeed in our automation. Being forced to use the same tools was a blessing in disguise in the end.

P.S. from Lisa Crispin: In one project (a good while ago), I knew I needed to get help from the developers to progress the automation, so I decided to use the same programming language for the tests that the developers were using. I bought myself a book and started writing scripts. When I needed help, the programmers were happy to help me because they knew the language.

Lesson

Cooperation between testers and developers is good for automation, and so is extending the tools you already use. And sometimes deviousness works!

29.13 An Automation Process for Large-Scale Success

Michael Snyman, South Africa

Test manager

I work for a large bank in South Africa, employing 25,000 staff. We adopted test automation from 2006 to 2008 with a clear aim of reducing costs and increasing productivity.

It was Edward Deming who said, “If you can’t describe what you are doing as a process, you don’t know what you are doing.” In our case, this was true; any success in the area of automation was due to individual skill and a large amount of luck. The challenge was in taking what the successful individuals did and describing this practice in the form of a process.

29.13.1 Where We Started

Our shelves were littered with numerous tool acquisitions and implementations with varying degrees of success. Each of these individual attempts had been focused on very limited and sometimes selfish objectives. The habit of looking only at accomplishing immediate project goals had significantly affected the ability of the organization to optimally use its selected tools. Such a one-sided view of automation had a considerable negative effect on operational activities such as regression testing and on justifying the investment made. Compounding the problem was the loss of valuable information in the form of test cases, test scenarios, and test data.

Lesson

Focusing on too low a level in the organization does not optimize automation as a whole.

Automation was involved too late in the process. How often is automation viewed as the savior of the behind-schedule project? When automation does not deliver on these unrealistic expectations, it becomes yet another automation failure. In reality, it is very different; my experience points to automation requiring multiple cycles and project releases for it to become fully effective and provide an acceptable ROI.

We weren't capitalizing on what we could have learned. For example, a failure experienced in production is an example of a test missed and one that should be included in the test cases for the next release. Test automation should provide an interface for both manual testers and incident management systems with the aim of capturing lessons learned during any phase in the project lifecycle.

The seeming lack of success in test automation initiatives and the large upfront investment required deters projects from planning and implementing test automation. The reluctance to learn from unsuccessful implementations and the habit of blaming the tool for failure in automation projects has resulted in a stigma linked to specific tools and automation in general.

Past attempts to justify automation focused on quality as the key attribute to be considered and measured. The difficulty in dealing with quality is that it is extremely complex. We clearly needed a way of providing a cost–benefit calculation for test automation using an attribute other than quality.

Good Point

Appropriate objectives are critical. Automation does not improve the quality of the tests or the software being tested.

In the absence of a detailed automation framework and process, a large dependency was placed on the skill and ability of individual team members.

29.13.2 The Key to Our Eventual Success: An Automation Process

In 2006, a formal project was launched with dedicated resources, a champion for automation, a good technical framework, clear goals, and a detailed plan. In this anecdote, I describe one aspect that was critical to our success in achieving automation on a large scale.

It was clear, based on past experience, that a standard approach for automation should be defined and documented in the form of a test automation process. However, this process could not exist in isolation but had to be integrated into the newly defined manual test process and should be compatible with the organizational software development lifecycle (SDLC). For example, in the requirement for a defined automation process, the framework required high-level activities described as *specification analysis*, *script creation*, *scenario documentation*, *validation*, and *data sourcing* that needed to be satisfied by a detailed process. The full process is shown in Figure 29.4.

Tip

The more people who are involved in automation, the better the documentation about it needs to be.

From the documented automation framework, we were able to extract the key process activities required to perform and support most automated testing activities. Here follows a brief description of the objective of each step.

- *Analysis and design:* Understand the client's requirements, and establish if it is possible to satisfy each requirement with current technology at our disposal.
- *Scripting and configuration:* Implement the client's requirements via an automated solution. This might include recoding, coding, and building special utilities.
- *Parameter definition:* Assess scripts against system user–defined scenarios with the aim of identifying elements to be parameterized.
- *Parameter management:* Manage large amounts of data in customized spreadsheets.
- *Scenario collection:* Populate spreadsheets with scenarios provided by stakeholders of the system.
- *Validation:* Check the spreadsheets and parameters, incorporating pass and fail criteria in the spreadsheets and allowing the automated script to validate results of executed tests.

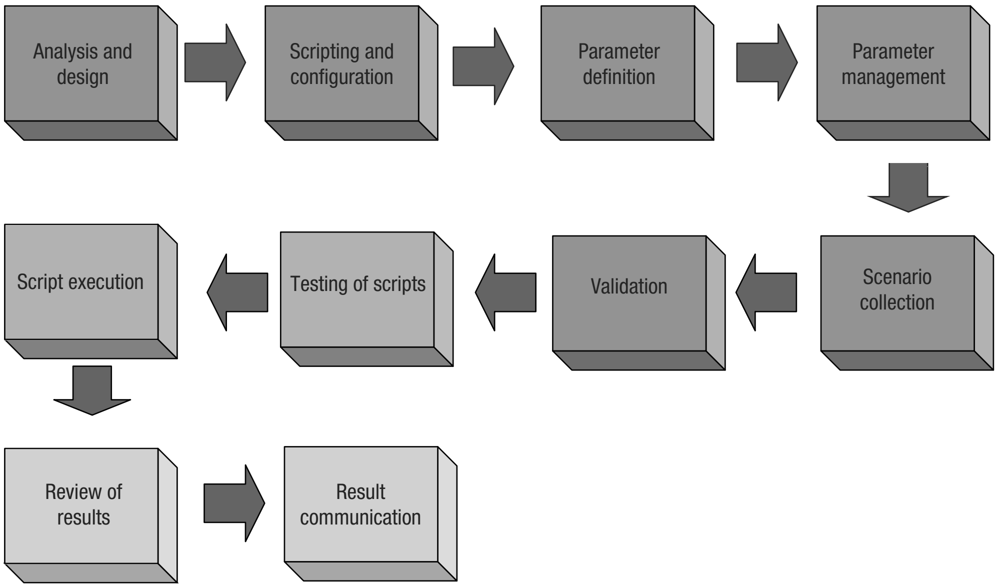

FIGURE 29.4 Automation process

- *Testing of scripts:* Ensure that the scripts run as expected, and remove any bugs in the scripts.
- *Script execution:* Run the scripts with the scenarios and parameters defined.
- *Review of results:* Internally review the results of script execution, what tests passed and failed, any common problems such as an unavailable environment, and so on.
- *Result communication:* Summarize the results sent to managers, developers, stakeholders, and others.

29.13.3 What We Learned

These are the main lessons we learned on our journey through test automation:

- Having a tool is not an automation strategy.
    a. The tool is nothing more than an enabler of a well-thought-out set of automation activities.
    b. We believe that if you approach automation correctly, you should be able to switch between tools with little or no impact.

- Automation does not test in the same way as manual testers do.
    a. Automation will never replace manual testers. We view automation as an extension of the manual tester, taking care of mundane activities such as regression testing, leaving the tester to get on with the more intellectual work.
- Record and playback is only the start.
    a. A set of recorded, unparameterized scripts has very limited reuse and ages quickly. The focus on data-driven automation provides us with the flexibility and reuse required.
- Automation test scripts are software programs and must be treated as such.
    a. Follow a standard software development life cycle in the creation of automated scripts.
    b. Document requirements; design, implement, and test your automated scripts.
- The value lies in the maintenance.
    a. The secret of getting a good return on your investment is reuse; for this to be possible, ensure maintenance is simple.
    b. Keyword or data-driven approach facilitates both reuse and easy maintenance.

29.13.4 Return on Investment

Our automation process enabled us to achieve consistency of automation practices across the bank. We showed a benefit of $8,600,000 after 3 years. This benefit calculation method was reviewed by our finance team at the highest level, and the benefits were confirmed by the individual system owner for whom the testing was done.

The total amount invested in the testing project, of which automation was a subproject, was in the area of $4,200,000. The amount spent on automation was less than 20 percent of this total budget, including the acquisition of functional testing tools, consulting, and the creation and execution of automated test scripts.

The benefit calculation was primarily based on the saving achieved in human resource costs. For example, one of our main systems used in the sales process took, on average, 4 weeks with 20 human resources to regression test. With automation, we reduced that process to 5 days and two resources: a reduction from 2,800 man-hours to 70 man-hours. This translated to a financial savings of about $120,500 per regression cycle. If you take into account that, on average, we run two full regression cycles per release and have multiple system releases per year, and that we are involved in various other systems, the savings soon start adding up.

We have a spreadsheet that uses parameters as the basis for all calculations. It allows us to compare the manual execution time per parameter to the automated time. We refer to parameters as the inputs required by the system under test (e.g., if we are testing a transfer from one account to another, parameters might be "from account," "to account," and "amount"). So, if we say that conservatively we invested $850,000 in automation and had benefit of $8,600,000, then the ROI for automation (ROI = (Gain – Cost)/Cost) was over 900 percent.

From a project testing perspective, the organization viewed the return on the total investment in testing, which was still over 100 percent. (Usually, if ROI is 10 percent or more, it is considered an excellent investment!)

It is also interesting to note that the automation part was the only initiative within the project testing that could be measured accurately, and as such, it provided justification for the entire project.

Good Point

Keep good records of the costs and benefits of automation (and testing) to make sure the highest levels in the organization realize what a good investment they have made in automating testing.

29.14 Test Automation Isn't Always What It Seems

Julian Harty, United Kingdom
Tester at large

I strongly believe testing can and should use automation appropriately, and conversely, we should be careful not to waste time and resources on automating garbage (e.g., ineffective, misguided, or useless tests). Also, we should beware of being beguiled by shiny automation for its own sake, and over the years, I've sadly met many people who believe, without foundation, that because they have automated tests, these tests are appropriate or sufficient. One of my self-assigned responsibilities as a test engineer is to challenge these flawed tests and retire as many as practical.

Good Point

Just because a test is automated doesn't make it a good test.

This anecdote includes several experience reports of test automation, both good and bad. Generally, I was directly involved in them, but sometimes the analysis was done by other team members. They are taken from companies I've worked with and for over the last 11 years. Project teams ranged from about 10 to 150 technical staff and typically ran for several years.

In every case, test automation was core to the project.

29.14.1 Just Catching Exceptions Does Not Make It a Good Test

A large global application included several APIs that allowed both internal and external groups to integrate with it. Java was the primary programming language. Over the years, before I was involved, hundreds of automated tests had been written for the respective APIs. For one API, the tests were written as a separate application, started from the command line, and in the other, the open source JUnit framework was used. Each set of tests ran slowly, and several days were required to update the tests after each release from the application's development team.

Our team of test engineers was asked to assume responsibility for both sets of tests. Each engineer was assigned to one set of tests. We spent several days learning how to simply run each set of tests (the process was cumbersome, poorly documented, and simply unreliable). We then started reading through the source code. What we found horrified us: There was an incredible amount of poorly written, duplicated code (implying little or no software design or structure), and worst of all, the only thing each test did to determine success or failure was catch runtime exceptions (e.g., out of memory, network timeout). When an exception was caught, the test reported a failure.

Good Point

Just because automated tests can be run doesn't make it good automation. You need to know the details of what the test does in order to assess whether or not it is a good test.

API tests should provide known inputs and confirm the results received are as expected without undesirable side effects or problems. For example, if we have an API for a calculator program, a typical method may be

```
result = divide(numerator, denominator);
```

A good test should check that the calculated result is within the error range for the sum (for real numbers, the answer may be approximated, truncated, or rounded, etc.). It should also check for what happens when invalid inputs (e.g., trying to divide by zero) are provided. For example, what should the result be, and should an exception be thrown? (And if so, which exception, and what should the exception contain?)

After spending several more weeks working on the test automation code, we ended up deleting all the tests in one case and effectively rewriting the tests for the other API. In both cases, we decided to focus on enhancing the lower-level unit tests written by the developers of the respective APIs rather than propagating or sustaining inadequate tests written by testing "specialists."

Good Point

Don't be afraid to throw away bad automation code and poor automated tests.

29.14.2 Sometimes the Failing Test Is the Test Worth Trusting

We decided to restructure our web browser–based tests because the existing tests had various problems and limitations, including high maintenance and poor reliability. The initial restructuring went well, and we also migrated from Selenium RC to WebDriver, which had a more compact and powerful API designed to make tests easier and faster to write. At this stage, the tests ran on a single machine, typically shared with the web application under test when run by the automated build process.

The tests took a long time to run (tens of minutes), which was much longer than our goal (of having them run within a few minutes). Thankfully, we had existing infrastructure to run the tests in parallel across banks of machines. The tests needed to connect to the appropriate test server, which was compiled and started by the build process, so the test engineer made what seemed to be the appropriate modifications to the automated tests to take advantage of the distributed testing infrastructure. Perplexingly, however, one of the tests failed every time he ran the tests using the distributed infrastructure.

Over the next few days, he dug into his code, the configuration scripts, and so on, but was unable to get the now embarrassingly obdurate test to pass. Finally, he discovered that a network configuration issue prevented any of the tests from reaching the newly built server; however, *only one of the tests detected this!* At this point, he was able to fix the network configuration issue and finally get the failing test to pass.

Good Point

Just because the tests pass, it doesn't mean that all is well. Tests need to be tested, too.

Several valuable lessons were learned:

- The other existing tests had effectively been worthless because they didn't fail when they could not reach the server at all.
- Even expert engineers can be fooled for days when test results don't conform to expectations.
- The failing test was actually the friend of the project because it exposed the problems with the rest of the—very flawed—tests.

One concept worth embracing is to consider how easily the current test could be fooled, or misled, into providing an erroneous result. For example, would an automated test for an email service detect missing menu options? Then consider how to strengthen the test so that it will not be fooled by this problem. While this concept can be applied iteratively to a given test, I suggest you limit yourself to addressing potentially significant problems; otherwise, your test code may take too long to write, maintain, and run.

29.14.3 Sometimes, Micro-Automation Delivers the Jackpot

In this story, 10 lines of Perl cracked open a critical nationwide system.

I learned many years ago that I'm not a brilliant typist. On one project, my poor typing helped expose a potential security issue when I accidentally mistyped some commands for a file transfer protocol in a telnet application, which led to a potential problem. Although I wanted to explore the potential flaw more scientifically, I continued to mistype commands in different ways and found that my mistakes were now hampering my ability to explore the application effectively. At the time, I lacked UNIX or Perl skills, so although writing an automated script to enter the commands seemed sensible, I was unsure whether it was worth spending the time to learn how to write a suitable script.

Salvation came in the form of a gnarly system administrator who knew both UNIX and Perl inside-out. Furthermore, the system architect had unilaterally decreed there were no security flaws in his file transfer protocol, and the system administrator saw a great opportunity to potentially prove the architect wrong, so he immediately offered to write a simple command-line script in Perl that would start telnet and issue various preliminary commands (those I'd been mistyping). The work took him less than 30 minutes.

Good Point

Don't hesitate to ask for help; it can save a lot of time (and hassle).

Once I had this basic script, I was able to experiment with the script through interactive typing, once the script had completed the initial steps, or by adding additional file transfer commands to custom versions of the script. With this script, we eventually proved that there were serious issues in the implementation of the file transfer protocol that resulted in buffer overflows in the underlying program, which we could then exploit to compromise the security of the system. I also identified several design flaws in the software update mechanism and proved that these flaws allowed an attacker to effectively disable the entire nationwide system. Not bad for a few hours work (and a few days to get permission to reproduce the problems in various environments, including the live production system).

INDEX