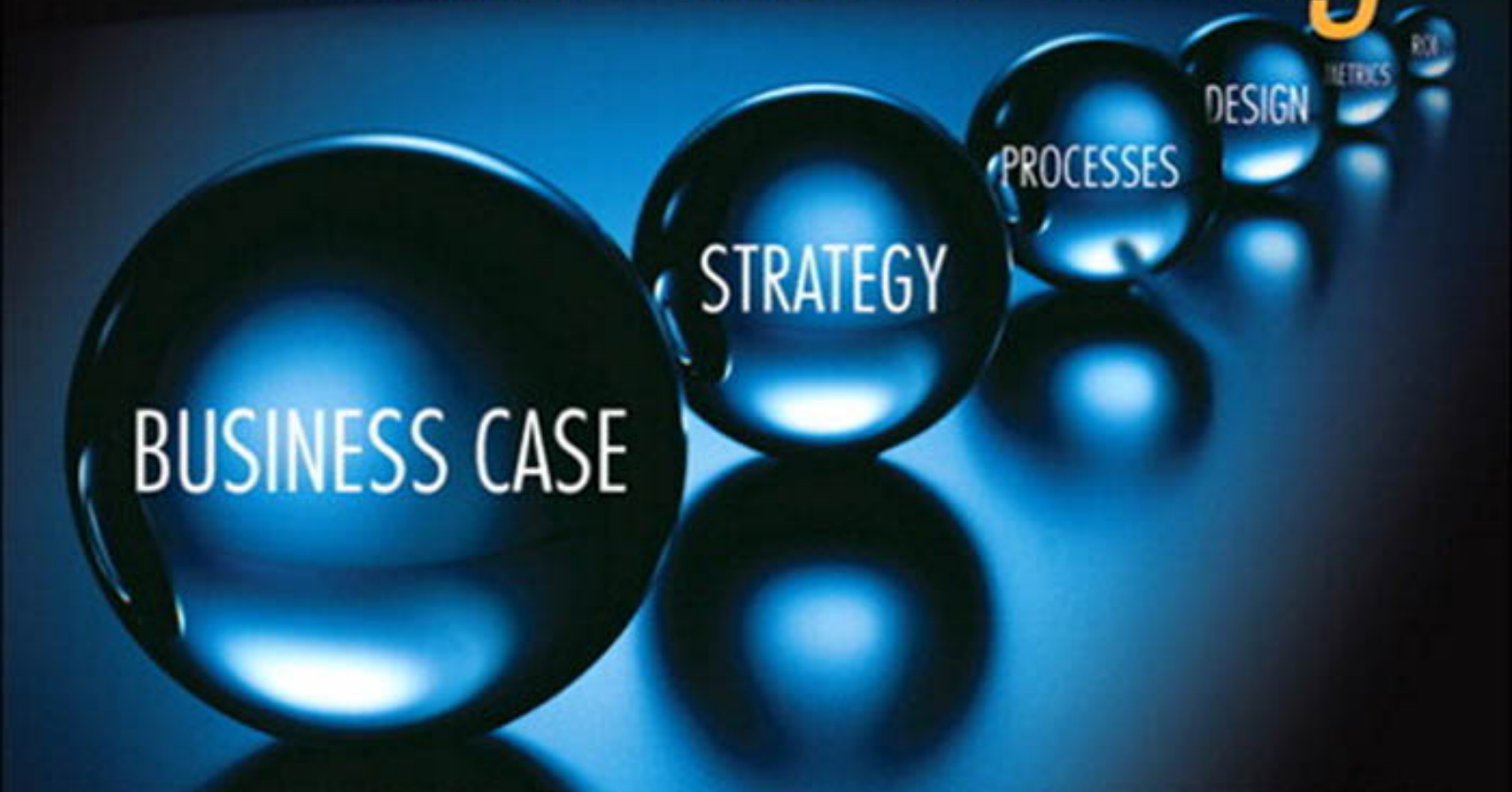
How To Save Time and Lower Costs While Raising Quality

Implementing Automated Software Testing



ELFRIEDE DUSTIN | THOM GARRETT | BERNIE GAUF

Forewords by **Admiral Edmund P. Giambastiani, Jr., U.S. Navy (Retired)** *and* **Dr. William C. Nylin, Jr.**

Library of Congress Cataloging-in-Publication Data

Dustin, Elfriede.
Implementing automated software testing : how to save time and lower costs while raising quality / Elfriede Dustin, Thom Garrett, Bernie Gauf.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-58051-6 (pbk. : alk. paper)
1. Computer software—Testing—Automation. I. Garrett, Thom. II. Gauf, Bernie. III. Title.

QA76.76.T48D8744 2009
005.1'4—dc22 2008055552

ISBN-13: 978- 0-321-58051-1
ISBN-10: 0-321-58051-6

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, March 2009

Foreword

by Admiral Edmund P. Giambastiani, Jr.

Today, the world turns over so rapidly that you have to build in a culture of change and innovation on a day-to-day basis. Innovation is an every-single-day part of a soldier, sailor, airman, Marine, or Coast Guardsman's life. Over the course of my military career, I was fortunate to see and experience the dramatic impact innovation has on the war fighter. One area where there has been tremendous innovation is in the field of information technology. The systems that we deploy today are now comprised of millions of lines of software, computer processor speeds we thought were unimaginable a decade ago, and networks that provide extraordinary bandwidth.

Despite these innovations, the need to respond to emerging threats is greater than ever, and the time in which we need to respond continues to decrease. From an information technology perspective, this means we need to be able to make software changes and field the associated capability improvements more rapidly than ever before. Rapidly but effectively testing changes is vital; however, for many programs more than 50% of the schedule is currently devoted to testing.

Innovative Defense Technologies (IDT) has taken the lead on providing an innovative solution to testing that I believe is needed in order for us to keep pace with an ever changing threat. With this book, *Implementing Automated Software Testing*, they have developed a guide that can help implement successful automated software testing programs and efforts. This book includes experience-based case studies and a thorough dissection of automated software testing issues and solutions. This book articulates how to develop the business case for automated software testing and provides a lifecycle approach to automated software testing programs. With *Implementing Automated Software Testing,* IDT is

providing timely and necessary material that allows responsible parties to implement an effective automated software testing program.

Admiral Edmund P. Giambastiani, Jr.
United States Navy (Retired)
Vice Chairman, Joint Chiefs of Staff (2005–2007)

Foreword

by Dr. William Nylin, Jr.

When I first began developing software systems in the mid-1960s, testing was primarily a programmer's responsibility; end users validated a relatively small sample of test cases. During the next two decades, more time was spent on testing, but there was significant test case overlap between programming staff testing and end users or a specific testing group. The redundant testing was expensive and delayed project implementation. When errors were discovered, the time lag for correction and revalidation was expensive. Today, a significant portion of the time and cost to market for software products is spent on testing. With the increasing complexity and size of the software included in products, my expectation is that the amount of testing required in the future will only continue to increase. Vast improvements in testing technologies are required, and automated software testing is one of the most promising answers.

The purpose of automated software testing is to increase testing efficiencies via effective use of time and resources, to allow for increased test permutations and combinations, as needed, to avoid test execution redundancy while increasing test coverage, and to allow for automated results analysis, resulting in increased quality and reliability of the software within the same or a reduced testing time frame. IDT's book *Implementing Automated Software Testing* provides extensive technical guidance for implementing an effective automated software testing program. The book provides experience-based automated software testing recommendations and solutions applicable to software testing programs across the board. Applying the automated software testing best practices and guidelines provided in this book will help improve your testing program and ultimately support your business in delivering software products on time, on budget, and with the highest quality. In addition, the book gives practical and

realistic advice on how to compute your return on investment for automated testing solutions. It helps the user understand where to best utilize automated testing and when it may not be cost-effective.

Finally, an additional advantage of automated software testing is the ability to formally audit the testing process. Section 404 of the Sarbanes-Oxley Act of 2002 (SOX 404) requires, as of 2004, that each annual report of a public company include a report by management on the company's internal control over financial reporting. Furthermore, the company's external auditors are required to attest to management's assessment. Management information systems are perhaps the most critical components of internal control systems. Thus, the ability to have an independent audit of the testing processes for new systems can be critical for future large-system development and implementation.

Dr. William Nylin, Jr.
Executive Vice Chairman and Director
Conn's, Inc.

Preface

Is your test automation strategy a losing proposition? Are you soured on the notion of automated software testing based on less than adequate past results? Are your test automation silver bullets missing their mark? Are you disappointed in your test automators? We at IDT[1] have identified a boilerplate solution, strategies, and ideas, all provided in this book, that can help increase the chances of your automated testing success.

Given the arsenal of system and application software testing strategies, techniques, and solutions, automated software testing is one of the most effective practices that if implemented correctly can help increase testing efficiencies and ultimately reduce the testing cost while contributing to increased systems and software quality in terms of faster, broader, and more efficient defect detection.

This book is a guide that can help organizations implement successful automated software testing programs and efforts. The book does not provide gimmicks or magical solutions, as none exist, but it provides experience-based discussions and recommendations. It includes a thorough dissection of automation issues, such as in Part I of the book, where we describe what automated software testing is and is not; why a business case is required for successful automation, including step-by-step instructions for developing one; why to automate and when. Then we summarize why automation often fails and the pitfalls and blunders that can be prevented; we describe the tools that are available to help implement successful automation efforts, with a focus on open-source testing tools. In Part II of the book we present six keys to successfully implementing automated software testing. These are

- Key 1: Know Your Requirements
- Key 2: Develop the Automated Test Strategy

1. www.idtus.com.

- Key 3: Test the Automated Software Test Framework (ASTF)
- Key 4: Continuously Track Progress—and Adjust Accordingly
- Key 5: Implement AST Processes
- Key 6: Put the Right People on the Project—Know the Skill Sets Required

IDT conducted two separate surveys related to automated software testing with approximately 700 total responses from test professionals all over the world, across organizations that were diverse in size and in what they do. The survey showed two very consistent themes:

- About 70% of survey respondents said they believe automation is high-payoff, but they are generally not sure why to automate and how automation applies to their project.
- Half of the survey respondents also said they felt they lacked the experience, time, or budgets to implement automation.

Most seem to agree: Automated software testing is useful, and an increasing need for it exists. However, the lack of experience seems to be the reason why automation is not implemented more often with a higher success rate. Finding people with the skills for the project is therefore important; a summary of skills required is provided in Chapter 10. For more details on the outcome of this survey, see Chapter 4.

Material Coverage and Book Organization

Part I: What Is Automated Software Testing and Why Should We Automate?

Chapter 1, What Is Effective Automated Software Testing (AST)?, describes what automated software testing is. The definition of *automated software testing* we use throughout this book is the "application and implementation of software technology throughout the entire software testing lifecycle (STL) with the goal to improve STL efficiencies and effectiveness."

In Chapter 2, Why Automate?, we address this question that is asked so often. Here we discuss the challenges of software testing today and how the time

and cost of software testing can be reduced. Reasons for why to automate, laying the foundation to help build the business case discussed step by step in Chapter 3, are presented here.

In Chapter 3, The Business Case, we define a step-by-step approach to defining the business case, which will cover the business need, the reasons for an automated software testing project, the business benefits (tangible and intangible), an analysis of the expected costs and timescales, an investment appraisal, and return on investment (ROI).

Chapter 4, Why Automated Software Testing Fails and Pitfalls to Avoid, clarifies some of the myths and realities surrounding automated software testing. The goal is for companies and organizations to review the lessons described here and not to repeat them during their automated software testing implementations.

Part II: How to Automate: Top Six Keys for Automation Payoff

Once management has been convinced by the business case that was laid out in Part I of this book and understands the pitfalls to avoid and the realities of automated testing, the next step is to determine how to automate. Part II of the book addresses how to successfully implement the various automated software testing tasks. We have determined that successful automated software testing can be achieved by implementing six top keys, described next.

Chapter 5, Key 1: Know Your Requirements, covers the importance of understanding the requirements before developing an automated testing strategy. Here we discuss approaches to determining the problem we are trying to solve along with how to gather information when requirements are not available.

Chapter 6, Key 2: Develop the Automated Test Strategy, discusses developing an automated testing approach in detailed steps, including test environment considerations, configuration management for automated test scripts, and related artifacts, among others. Here we also discuss what to consider when deciding what to automate and the importance of choosing the right tool, whether open-source, vendor-provided, or in-house-developed.

Chapter 7, Key 3: Test the Automated Software Test Framework (ASTF), covers the importance of understanding testing techniques and documenting test cases as part of automated testing. Automators often forget that documentation is still a vital part of the automated test program. The test case documentation serves as the blueprint for the automated software testing efforts. This chapter describes the importance of tracing test cases back to requirements; the content of the test cases, such as needing to include inputs and expected results;

and how documented test cases become the basis for developing and implementing the automated tests.

Chapter 8, Key 4: Continuously Track Progress—and Adjust Accordingly, addresses the importance of tracking the goal that was set at the outset of the automation program. For example, during the discussion of business case development in Chapter 3 we explain the need for defining goals; in this chapter we discuss how peer reviews, inspections, and various automation and testing metrics can help measure and track progress against those goals.

Chapter 9, Key 5: Implement AST Processes, points out the need for a lightweight process. Some automated testing scripts can be implemented successfully without much process in place, but in order to effectively implement a large automated testing program a lightweight adaptable process should be in place. This chapter discusses a summary of this process, linking back to the details in various chapters.

Chapter 10, Key 6: Put the Right People on the Project—Know the Skill Sets Required, clarifies the skill sets needed for developing automated software testing, for instance, a skill set similar to that of the software development team, which includes requirements analysis, design, software development, and testing. Key 6 points out that although knowledge of testing techniques and analytical skills is important, effective automated software testing implementation requires software development skills. The skills described here parallel the automated testing process described in Chapter 9.

Audience

The target audience of this book is software test professionals such as test managers, leads, and practitioners. It is also geared toward all quality assurance professionals, QA leads, and practitioners. Project managers and software developers looking to improve the effectiveness and quality of their software delivery will also benefit from this book.

Chapter 4

Why Automated Software Testing Fails and Pitfalls to Avoid

Approaching AST in a realistic manner

Even though most companies believe that AST is useful, few companies can claim actual success. Various user group postings and a comprehensive survey conducted by IDT over the course of one year in 2007 support this finding.

In this chapter we will analyze the issue of why so many AST efforts fail and how to avoid some of the pitfalls, and we will clarify some of the misperceptions surrounding AST. Chapter 9 further explains why our proposed lightweight process based on the automated test lifecycle methodology (ATLM) already described in *Automated Software Testing*[1] will help solve many of the current AST woes.

A year-long IDT automated software testing survey was conducted; it was posted on commercial QA user group sites, sent to tens of thousands of test engineers, posted on government tech sites such as Government Computer News and Defense Systems, and announced during a webinar[2] we conducted called "Automated Testing Selected Best Practices." We received over 700 responses, worldwide. Here is a breakdown of the respondents' demographics:

- Over 73% of the respondents were from the United States, and the rest were from India, Pakistan, China, Europe, and other locations throughout the world.
- Nearly 70% identified their organization type as commercial, 10% claimed government direct employment, and the rest "other," such as governmental contractors, educational, or independent.
- For 40% the size of the organization was less than or equal to 300 employees; 60% claimed an organization size of more than 300 employees.

The outcome of the survey showed that the value of AST is generally understood, but often automation is not used or it fails. In the survey we asked respondents why in their experience automation is not used, and the largest percentage responded that AST does not get implemented because of lack of resources—time, budget, skills.

Closely related to these questions, we also received feedback as to why automation fails. The highest percentage responded that many AST efforts fail and the tools end up as shelfware for reasons similar to those for why it's not used in the first place The reasons given for failure of AST were

- Lack of time: 37%
- Lack of budget: 17%
- Tool incompatibility: 11%
- Lack of expertise: 20%
- Other (mix of the above, etc.): 15%

In summary, although 72% stated that automation is useful and management agrees, they either had not implemented it at all or had had only limited success.

Here are various quotes providing reasons for limited AST success or failure:

- "We have begun implementing, but aren't allowed significant time to do so."
- "Have implemented some efforts but lack of time, budget, and resources prohibit us to fully perform this function."
- "The company has previously implemented automated testing successfully, but this was years ago and we currently don't have the time or budget to reimplement."

- "I'm the only one automating (so have some automation), but spend too much time on new feature release, need more people."
- "Accuracy of automated processes [is] the largest issue we have encountered."

Our survey results match what our experience has shown over the years: Many agree that AST is the best way to approach testing in general, but there is often a lack of budget, time, or experience available to execute it successfully.

Additional reasons why AST fails include the following:

- R&D does not generally focus on testing (manual or automated).
- Myths and misperceptions about AST persist.
- There is a lack of AST processes.
- There is a lack of software development considerations for AST.
- There is a lack of AST standards.

Each of these issues is further discussed in the next sections.

4.1 R&D Does Not Generally Focus on Automated or Manual Testing Efforts

R&D and its resulting technologies have been fueling high-tech product innovation over the last 20 to 30 years. Yet our ability to test these technologies has not kept pace with our ability to create them. In our experience, most reports on technology advances cover only R&D, and not related test technologies. For example, *Business Week* reported on "The World's Most Innovative Companies" in an April 17, 2008, article.[3] As so often happens, the article focuses on R&D only, but no attention is paid to research and development and test (R&D&T), i.e., how should this great technology be tested? Innovation does not seem to consider related required testing technologies; testing, however, has become more important than ever.

We have spent much time researching the latest testing technologies, and based on our research we have come up with the "Automatically Automate Your Automation" (AAA) concept,[4] described in Chapter 1 but also here—see "GUI Testing Recommendations" for additional examples of this concept—and

throughout this book. We are currently further refining the concept of AAA, but we also have determined some interesting trends that need to be dealt with, which include the following:

- **Software development and testing are driving the business.**

  Business needs almost exclusively used to drive software testing technologies, but the trend is shifting in both directions: Software testing is now also driving the business. Business executives can have the best business ideas, but if the software and testing efforts lag behind and/or the system or product is delivered late and/or with low quality, the competition is only a few clicks away. First to market with high quality is key.

- **Attention should be paid to "perceived" versus "actual" quality.**

  The best-quality processes and standards cannot solve the perception issue. For example, ten defects that occur very frequently and impact critical functionality would be *perceived* by any customer as poor quality, even if the defect density was very low. On the other hand, 100 defects that occur very infrequently and have almost no impact on operations would usually be *perceived* by an end user as good quality, even if the defect density was high. Not much research goes into "usage-based testing," which exploits the concept of perceived quality, yielding higher perceived quality, and thus happier customers. One such example of great perceived quality and usability is Amazon.com versus all other online booksellers—in our experience, Amazon.com is just more user-friendly; they are, as *Business Week* refers to them, an "e-tail maverick."[5]

  The goal needs to be improving perceived quality. This can be accomplished by focusing testing on the usability of the most often used functionality (which absolutely has to work without defects), and on reliability (the probability that no failure will occur in the next *n* time intervals. (See Section 2.3 in Chapter 2 for more detail on MTTF.)

- **Testing invariably gets some of the blame, if not all.**

  Deadlines are looming and the testing cycle in multiple environments can be numerous and seemingly endless. Testing often gets blamed for missed deadlines, projects that are over budget, uncovered production defects, and lack of innovation. But often the real culprits are inefficient systems engineering processes, such as the black-box approach where millions of software lines of code (SLOC) are developed, including vast

amounts of functionality, only to be handed over to a test team so they can test and peel back the layers of code, painstakingly finding one defect after another, sometimes not uncovering a major showstopper until another defect is fixed. In summary, some of the real culprits of testing being late are

- Poor development practices, resulting in buggy code, requiring long and repetitive fixing cycles; these include
  - Lack of unit testing. Statistics show (and our experience backs them up) that the more effective the unit testing efforts are, the smoother and shorter the system testing efforts will be.
  - Inefficient build practices. Build and release processes should be automated. If they are not, building software can be time-consuming and error-prone.
  - Inefficient integration or system tests by the development teams (see "Developers don't system-test" below).
- Unrealistic deadlines. Often deadlines are set in stone without much consideration for how long it will actually take to develop or test particular software. Poor estimating practices do not typically include or give credence to test team estimates, or development estimates are not acceptable to those who have overpromised the product. Setting unrealistic deadlines is a sure way of setting up deliverables for failure.
- Also see Section 3.4, Risks, for other culprits.

- **Developers don't system-test.**

  Although many developers conduct unit testing, and proponents of test-driven software development generally do a good job of testing their software modules, there is still a lack of developer integration or system testing. Some might suggest shifting away from the testing focus and instead focusing on improving development processes. This is not a bad idea, but even with the best processes and the most brilliant developers in-house, software development is an art—integration and system testing will always be required. Human factors influence why developers don't system-test: They generally don't have time; or they don't specialize in testing and testing techniques; and often they are busy churning out new code and functionality. Developers are strapped cranking out new features while trying to meet unreasonable deadlines. First to market, again, is the key.

In summary, technology research groups often focus only on R&D, but the focus should rather be on R&D&T, i.e., how to test these latest and greatest inventions.

4.2 AST Myths and Realities

When we originally wrote the book *Automated Software Testing,* we tried to clarify many of the misperceptions and myths about AST. The book was published in 1999. Now, ten years later, unfortunately, our experience has shown and our survey backs up the fact that many of these same myths still exist.[6]

Let's say that while you're at a new project kickoff meeting, the project manager introduces you as the test lead. The project manager mentions that the project will use an automated test tool and adds that because of this the test effort is not expected to be significant. The project manager concludes by requesting that you submit within the next week a recommendation of the specific test tool required, together with a cost estimate for the procurement of the tool. You are caught by surprise by the project manager's remarks and wonder about the manager's expectations with regard to automated testing. Any false automated testing expectations need to be cleared up immediately. Along with the idea of automated testing come high expectations. A lot is demanded from technology and automation. Some people have the notion that an automated test tool should be able to accomplish everything from test planning to test execution, without much manual intervention. Although it would be great if such a tool existed, there is no such capability on the market today. Others believe that it takes only one test tool to support all test requirements, regardless of environmental parameters such as the operating system or programming language used.

Some may incorrectly assume that an automated test tool will immediately reduce the test effort and the test schedule. It has been proven successfully that automated testing is valuable and can produce a return on investment, but there isn't always an immediate payback on investment. This section addresses some of the misconceptions that persist in the software industry and provides guidelines for how to manage some of the automated testing misconceptions.

Automatic Test Plan Generation

Currently, there is no commercially available tool that can create a comprehensive test plan while also supporting test design and execution.

Throughout a software test career, the test engineer can expect to witness test tool demonstrations and review an abundant amount of test tool literature. Often the test engineer will be asked to stand before a senior manager or a small number of managers to give a test tool functionality overview. As always, the presenter must bear in mind the audience. In this case, the audience may consist of individuals with just enough technical knowledge to make them enthusiastic about automated testing, but they may be unaware of the complexity involved with an automated test effort. Specifically, the managers may have obtained information about automated test tools third-hand and may have misinterpreted the actual capability of automated test tools.

What the audience at the management presentation may be waiting to hear is that the tool that you are proposing automatically develops the test plan, designs and creates the test procedures, executes all the test procedures, and analyzes the results for you. You meanwhile start out the presentation by informing the group that automated test tools should be viewed as enhancements to manual testing, and that automated test tools will not automatically develop the test plan, design and create the test procedures, and execute the test procedures. AST does not replace the analytical thought required to develop a test strategy or the test techniques needed to develop the most effective tests.

Not far into the presentation, and after several management questions, it becomes apparent just how much of a divide exists between the reality of the test tool's capabilities and the perceptions of the individuals in the audience. The term *automated test tool* seems to bring with it a great deal of wishful thinking that is not closely aligned with reality. An automated test tool will not replace the human factor necessary for testing a product. The proficiencies of test engineers and other quality assurance experts are still needed to keep the testing machinery running. AST can be viewed as an additional part of the machinery that supports the release of a good product, and only after careful consideration and effective implementation will it yield success.

Test Tool Fits All

Currently not one single test tool exists that can be used to support all operating system environments.

Generally, a single test tool will not fulfill all the testing requirements of an organization. Consider the experience of one test engineer encountering such a situation. The test engineer was asked by a manager to find a test tool that could be used to automate all real-time embedded system tests. The department was

using VxWorks and Integrity, plus Linux and Windows XP, programming languages such as Java and C++, and various servers and Web technologies.

Expectations have to be managed, and it has to be made clear that currently there is no one single tool on the market that is compatible with all operating systems and programming languages. Often more than one tool and AST technique are required to test the various AUT technologies and features (GUI, database, messages, etc.).

Immediate Test Effort Reduction

Introduction of automated test tools will not immediately reduce the test effort. A primary impetus for introducing an automated test tool into a project is to reduce the test effort. Experience has shown that a learning curve is associated with the attempts to apply automated testing to a new project and the effective use of automated testing. Test effort savings do not necessarily come immediately. Still, test or project managers have read the test tool literature and are anxious to realize the potential of the automated tools.

Surprising as it may seem, our experience has shown that the test effort actually increases initially when an automated test effort is first introduced into an organization. When introducing automated testing efforts, whether they include a vendor-provided test tool or an open-source, freeware, or in-house-developed solution, a whole new level of complexity and a new way of doing testing are being added to the test program. And although there may be a learning curve for the test engineers to become smart and efficient in the use of the tool, there are still manual tests to be performed on the project. Additionally, new skills might be required, if, for example, the goal is to develop an automated test framework from scratch or to expand on an existing open-source offering. The reasons why an entire test effort generally cannot be automated are outlined later in this chapter.

Initial introduction of automated testing also requires careful analysis of the AUT in order to determine which sections of the application can be automated (see "Universal Application of AST" for further discussion). Test automation also requires careful attention to automated test procedure design and development. The automated test effort can be viewed as a mini development lifecycle, complete with the planning and coordination issues that come along with a development effort. Introducing an automated test tool requires that the test team perform the additional activities as part of the process outlined in Chapter 9.

Immediate Reduction in Schedule

An automated test tool will not immediately minimize the testing schedule.

Another automated test misconception is the expectation that the introduction of automated tests in a new project will immediately minimize the test schedule. Since the testing effort actually increases when an automated testing effort is initially introduced, as previously described, the testing schedule will not experience the anticipated decrease at first. This is because in order to effectively implement AST, a modified testing process has to be considered, developed, and implemented. The entire test team and possibly the development team need to be familiar with this modified effort, including the automated testing process, and need to follow it. Once an automated testing process has been established and effectively implemented, the project can expect to experience gains in productivity and turnaround time that have a positive effect on schedule and cost.

Tool Ease of Use

An automated tool requires new skills; therefore, additional training is required. Plan for training and a learning curve!

Many tool vendors try to sell their tools by exaggerating the *ease of use* of the tool and deny any learning curve associated with it. The vendors are quick to point out that the tool can simply capture (record) a test engineer's keystrokes and (like magic) create a script in the background, which then can simply be reused for playback. Efficient automation is not that simple. The test scripts that the tool automatically generates during recording need to be *programmatically* modified manually, requiring tool scripting and programming knowledge, in order to make the scripts robust and reusable (see "Equating Capture/Playback to AST" for additional detail). Scripts also need to be modified in order for them to become maintainable. A test engineer needs to be trained on the tool and the tool's built-in scripting language to be able to modify the scripts, or a developer needs to be hired in order to use the tool effectively. New training or hiring requirements and/or a learning curve can be expected with the use of any new tool. See Chapter 10 on skills required for AST.

Universal Application of AST

Not all tests can be automated.

As discussed previously, automated testing is an enhancement to manual testing. Therefore, and along with other reasons already articulated in this book, it is

unreasonable to expect that 100% of the tests on a project can be automated. For example, when an automated GUI test tool is first introduced, it is beneficial to conduct some compatibility tests on the AUT in order to see whether the tool will be able to recognize all objects and third-party controls.

The performance of compatibility tests is especially important for GUI test tools, because such tools have difficulty recognizing some custom controls features within the application. These include the little calendars or spin controls that are incorporated into many applications, especially Web or Windows applications. These controls or widgets are often written by third parties, and most test tool manufacturers can't keep up with the hundreds of clever controls churned out by various companies. The grid objects and embedded objects within the various controls are very popular and much used by developers; however, they can be challenging for test automators, as the tool they are using might not be compatible with control *x*.

It may be that the test tool is compatible with all releases of C++ and Java, for example, but if an incompatible third-party custom control is introduced into the application, the tool may not recognize the object on the screen. It may be that most of the application uses a third-party grid that the test tool does not recognize. The test engineer will have to decide whether to automate this part of the application, for example, by defining a custom object within the automation tool; find another work-around solution; or testing the control manually.

Other tests are impossible to automate completely, such as verifying a printout. The test engineer can automatically send a document to the printer but then has to verify the results by physically walking over to the printer to make sure the document really printed. The printer could be off-line or out of paper. Certainly an error message could be displayed by the system—"Printer off-line" or "Printer out of paper"—but if the test needs to verify that the messages are accurate, some physical intervention is required. This is another example of why not every test can be automated.

Often associated with the idea that an automated test tool will immediately reduce the testing effort is the fallacy that a test tool will be able to automate 100% of the test requirements of any given test effort. Given an endless number of permutations and combinations of system and user actions possible with *n*-tier (client/middle layer/server) architecture and GUI-based applications, a test engineer or team does not have enough time to test every possibility, manually or automated (see "100% Test Coverage," next).

Needless to say, the test team will not have enough time or resources to support 100% test automation of an entire application. It is not possible to test

all inputs or all combinations and permutations of all inputs. It is impossible to test exhaustively all paths of even a moderate system. As a result, it is not feasible to approach the test effort for the entire AUT with the goal of testing 100% of the software application. (See 100% Test Coverage, below.)

Another limiting factor is cost. Some tests can be more expensive to automate than to execute manually. A test that is executed only once is often not worth automating. For example, an end-of-year report of a health claim system might be run only once, because of all the setup activity involved to generate the report. Since this report is executed rarely, automating it may not pay off. When deciding which test procedures to automate, a test engineer needs to evaluate the value or payoff of investing time in developing an automated script.

The test team should perform a careful analysis of the application when selecting the test requirements that warrant automation and those that should be executed manually. When performing this analysis, the test engineer will also need to weed out redundant tests. The goal for test procedure coverage, using automated testing, is for each test to exercise multiple items, while avoiding duplication of tests. For each test, an evaluation should be performed to ascertain the value of automating it. This is covered in Chapter 6.

100% Test Coverage

Even with automation, not everything can be tested.

One of the major reasons why testing has the potential to be an infinite task is that in order to know that there are no problems with a function, it must be tested with all possible data, valid and invalid. Automated testing may increase the breadth and depth of test coverage, yet with automated testing there still isn't enough time or resources to perform a 100% exhaustive test.

It is impossible to perform a 100% test of all the possible simple inputs to a system. The sheer volume of permutations and combinations is simply too staggering. Take, for example, the test of a function that handles the verification of a user password. Each user on a computer system has a password, which is generally six to eight characters long, where each character is an uppercase letter or a digit. Each password must contain at least one digit. How many possible character combinations do you think there are in this example? According to Kenneth H. Rosen in *Discrete Mathematics and Its Applications*[7], there are 2,684,483,063,360 possible variations of passwords. Even if it were possible to create a test procedure each minute, or 60 test procedures per hour, equaling 480 test procedures per day, it would still take 155 years to prepare and execute

a complete test. Therefore, not all possible inputs can be exercised during a test. With this rapid expansion it would be nearly impossible to exercise all inputs, and in fact it has been proven to be impossible in general.

It is also impossible to exhaustively test every combination and path of a system. Let's take, for example, a test of the telephone system in North America. The format of telephone numbers in North America is specified by a numbering plan. A telephone number consists of ten digits, which are split into a three-digit area code, a three-digit office code, and a four-digit station code. Because of signaling considerations, there are certain restrictions on some of these digits. A quick calculation shows that in this example 6,400,000,000 different numbers are available—and this is only the valid numbers; we haven't even touched on the invalid numbers that could be applied. This is another example that shows how it is impractical, based upon development costs versus ROI, to test all combinations of input data for a system and that various testing techniques need to be applied to narrow down the test data set.[8]

The preceding paragraphs outlined why testing is potentially an infinite task. In view of this, code reviews of critical modules are often done. It is also necessary to rely on the testing process to discover defects early. Such test activities, which include requirement, design, and code walk-throughs, support the process of defect prevention. Both defect prevention and detection technologies are discussed further throughout this book. Given the potential magnitude of any test, the test team needs to rely on test procedure design techniques, such as equivalence testing, where only representative data samples are used. See Chapter 6 and throughout this book for references to this technique.

Equating Capture/Playback to AST

Hitting a Record button doesn't produce an effective automated script. Many companies and automated testers still equate AST with simply using capture/playback tools. Capture/playback in itself is inefficient AST at best, creating nonreusable scripts at worst.

Capture/playback tools record the test engineer's keystrokes in some type of scripting language and allow for script playback for baseline verification. Automated test tools mimic the actions of the test engineer. During testing, the engineer uses the keyboard and mouse to perform some type of test or action. The testing tool captures all keystrokes and subsequent results, which are recorded and baselined in an automated test script. During test playback, scripts compare the latest outputs with the baseline. Testing tools often provide built-

in, reusable test functions, which can be very useful, and most test tools provide for nonintrusive testing; i.e., they interact with the AUT as if the test tool were not involved and won't modify/profile code used in the AUT.

Yet capture/playback-generated scripts provide many challenges. For example, the capture/playback records hard-coded values; i.e., if you record a data input called "First Name," that "First Name" will be hard-coded, rendering test scripts usable only for that "First Name." If you wanted to read in more than one "First Name," you would have to add the capability of reading data from a file or database, and include conditional statements and looping constructs. Variables and various functions have to be added and scripts need to be modified—i.e., software development best practices have to be applied—to make them effective, modular, and repeatable.

Additionally, capture/playback doesn't implement software development best practices right out of the box; scripts need to be modified to be maintainable and modular. Also, vendor-provided capture/playback tools don't necessarily provide all testing features required; code enhancements are often necessary to meet testing needs. Finally, vendor-provided tools are not necessarily compatible with the systems engineering environment, and software testing scripts need to be developed in-house.

AST Is a Manual Tester Activity

Capture/playback tool use and script recording do not an automated tester make.

Although AST does not replace the manual testing and analytical skills required for effective and efficient test case development, the AST skills required are different from manual software testing skills. Often, however, companies buy into the vendor and marketing hype that AST is simply hitting a Record button to automate a script. However, capture/playback tool use does not an automated tester make.

It is important to distinguish the skills required for manual software testing from those required for automated software testing: Mainly, an automated software tester needs software development skills. What is required from a good automated software tester is covered in detail in Chapter 10. The important point to note here is that different skills are required, and a manual tester without any training or background in software development will have a difficult time implementing successful automated testing programs.

Losing Sight of the Testing Goal: Finding Defects

The AST goal is to help improve quality, not to duplicate development efforts.

Often during AST activities the focus is on creating the best automation framework and the best automation software, and we lose sight of the testing goal, i.e., to find defects. As mentioned, it's important that testing techniques, such as boundary value testing, risk-based testing, and equivalence partitioning, are being used to derive the best suitable test cases.

You may have employed the latest and greatest automated software development techniques and used the best developers to implement your automation framework. It performs fast with tens of thousand of test case results, efficient automated analysis, and great reporting features. It is getting rave reviews. But no matter how sophisticated your automated testing framework is, if defects slip into production and the automated testing scripts didn't catch the defects it was supposed to, your automated testing effort will be considered a failure.

It's therefore important to conduct a metrics assessment similar to the one discussed in the ROI section of Chapter 3 that will allow you to determine whether the automated testing effort is finding the defects it should.

Focusing on System Test Automation and Not Automating Unit Tests

Automating unit tests can contribute to the success of all later test activities.

Our experience has shown that if unit testing is automated, along with automated integration test and build processes, subsequent system testing activities uncover fewer defects, and the system testing lifecycle can be reduced. Another benefit of automated unit testing is that the sooner in the STL a defect is uncovered, the cheaper it is to fix; i.e., much less effort is involved in fixing a unit test defect affecting one unit or even fixing an integration test defect affecting some components than in finding and fixing a defect during system testing, when it could affect various system components, making analysis cumbersome, or even affect other parts of the STL effort, in the worst case pointing back to an incorrectly implemented requirement.

Additionally, software evolution and reuse are very important reasons for automating tests. For example, ideally, automated unit, component, and integration tests can be reused during system testing. Automating a test is costly and may not be justified if the test is going to be run only once or a few times. But if

tests are run dozens of times, or hundreds of times, in nightly builds, and rerun during different testing phases and various configuration installs, the small cost of automation is amortized over all those executions.[9]

4.3 Lack of Software Development Considerations for AST

AST efforts can fail when software development doesn't take into account the automated testing technologies or framework in place. Software developers can contribute to the success of automated testing efforts if they consider the impacts on them when making code or technology changes. Additionally, if developers consider some of the selected best practices described here, AST efforts can reap the benefits. The selected best practices include the following:

- Build testability into the application.
- Facilitate automation tool recognition of objects: Uniquely name all objects, considering various platforms—client/server, Web, etc.—and GUI/interface testing considerations, such as in the case of Windows development, for example, within the Windows architecture. Additionally, don't change the object names without AST considerations; see also "GUI Object Naming Standards" later in this chapter.
- Follow standard development practices; for example, maintain a consistent tab sequence.
- Follow best practices, such as the library concept of code reuse, i.e., reusing existing already tested components, as applicable (discussed in Chapter 1).
- Adhere to documentation standards to include standard ways of documenting test cases and using the OMG[10] IDL, for example, which would allow for automated test case code generation (further defined in "Use of OMG's IDL" later in this chapter).
- Adhere to the various standards discussed later on in this chapter, such as Open Architecture standards, coding standards, and so forth.

Each of these recommendations is discussed in more detail in the following sections.

Build Testability into the Application

Software developers can support the automated testing effort by building testability into their applications, which can be supported in various ways. One of the most common ways to increase the testability of an application is to provide a logging, or tracing, mechanism that provides information about what components are doing, including the data they are operating on, and any information about application state or errors that are encountered while the application is running. Test engineers can use this information to determine where errors are occurring in the system, or to track the processing flow during the execution of a test procedure.

As the application is executing, all components will write log entries detailing what methods, also known as functions, they are currently executing and the major objects they are dealing with. The entries are written typically to a disk file or database, properly formatted for analysis or debugging, which will occur at some point in the future, after the execution of one or more test procedures. In a complex client-server or Web system, log files may be written on several machines, so it is important that the log include enough information to determine the path of execution between machines.

It is important to place enough information into the log that it will be useful for analysis and debugging, but not so much information that the overwhelming volume will make it difficult to isolate important entries. A log entry is simply a formatted message that contains key information that can be used during analysis. A well-formed log entry includes the following pieces of information:

- **Class name and method name:** This can also simply be a function name if the function is not a member of any class. This is important for determining a path of execution through several components.
- **Host name and process ID:** This will allow log entries to be compared and tracked if they happen on different machines or in different processes on the same machine.
- **Timestamp of the entry (to the millisecond at least):** An accurate timestamp on all entries will allow the events to be lined up if they occur in parallel or on different machines.
- **Messages:** One of the most important pieces of the entry is the message. It is a description, written by the developer, of what is currently happening in the application. A message can also be an error encountered during execution, or a result code from an operation. Gray-box testing will

greatly benefit from the logging of persistent entity IDs or keys of major domain objects. This will allow objects to be tracked through the system during execution of a test procedure. See also gray box testing, described in Chapter 1.

Having these items written to the log file by every method, or function, of every component in the system can realize the following benefits:

- The execution of a test procedure can be traced through the system and lined up with the data in the database that it is operating on.
- In the case of a serious failure, the log records will indicate the responsible component.
- In the case of a computational error, the log file will contain all of the components that participated in the execution of the test procedure and the IDs or keys of all entities used.
- Along with the entity data from the database, this should be enough information for the test team to pass on to the development personnel who can isolate the error in the source code.

Following is an example of a log file from an application that is retrieving a customer object from a database:

```
Function:   main (main.cpp, line 100)
Machine:    testsrvr (PID=2201)
Timestamp:  3/10/2009 20:26:54.721
Message:    connecting to database [dbserver1,
            customer_db]

Function:   main (main.cpp, line 125)
Machine:    testsrvr (PID=2201)
Timestamp:  3/10/2009 20:26:56.153
Message:    successfully connected to database [dbserver1,
            customer_db]

Function:   retrieveCustomer (customer.cpp line 20)
Machine:    testsrvr (PID=2201)
Timestamp:  3/10/2009 20:26:56.568
Message:    attempting to retrieve customer record for
            customer ID [A1000723]
```

```
Function:    retrieveCustomer (customer.cpp line 25)
Machine:     testsrvr (PID=2201)
Timestamp:   3/10/2009 20:26:57.12
Message:     ERROR: failed to retrieve customer record,
             message [customer record for ID A1000723 not
             found]
```

This log file excerpt demonstrates a few of the major points of application logging that can be used for effective testing.

- In each entry, the function name is indicated, along with the filename and line number in the code where the entry was written. The host and process ID are also recorded, as well as the time that the entry was written.
- Each message contains some useful information about the activity being performed; for example, the database server is `dbserver1`, the database is `customer_db`, and the customer ID is `A1000723`.
- From this log it is evident that the application was not able to successfully retrieve the specified customer record.

In this situation, a tester could examine the database on `dbserver1`, using SQL tools, and query the `customer_db` database for the customer record with ID `A1000723` to verify its presence. This information adds a substantial amount of defect diagnosis capability to the testing effort, since the tester can now pass this information along to the development staff as part of the defect information. The tester is now not only reporting a "symptom," but along with the symptom can also document the internal application behavior that pinpoints the cause of the problem.

Adhere to Open Architecture Standards

Open Architecture (OA) principles, described in the Open Architecture Computing Environment Design Guidance and in the Open Architecture Computing Environment Technologies and Standards documents,[11] were developed by the U.S. Navy and emphasize the use of widely adopted industry standards and component-based technologies. The open standards approach has been demonstrated to reduce cost and improve rapid insertion of new software capability into an existing system.

By implementing and following the OA standards, developers can expect various benefits, including assured technical performance, reduced lifecycle cost, affordable technology refresh, and reduced upgrade cycle time. Additional expected benefits include

- Scalable, load-invariant performance
- Enhanced information access and interoperability
- Enhanced system flexibility for accomplishment of mission and operational objectives
- Enhanced survivability and availability
- Reduced lifecycle cost and affordable technology refresh
- Reduced cycle time for changes and upgrades

The Defense Advanced Research Projects Agency (DARPA), academia, and industry's R&D efforts have focused on certain architectural concepts intended to foster lifecycle cost benefits, as well as technical performance benefits. Developing software using OA will result in additional benefits, such as

- Open architectures
- Distributed processing
- Portability
- Scalability
- Modularity
- Fault tolerance
- Shared resource management
- Self-instrumentation

For additional details and best development practices specific to automated testing tool development, see Table 8-5, "Test Development Guidelines," in the book *Automated Software Testing*.[12]

Adhere to Standard Documentation Format

Software testing efforts often include sifting through documentation to verify that all information is provided. Documentation assessment efforts can also be

automated, but we would like to offer the following recommendation for software developers or documentation teams in order to support their successful automation: Currently almost all software providers and vendors use various documentation formats to produce documentation deliverables; no one specific format is being followed. We recommend that a standard documentation format be used, i.e., templates that offer multiple-choice offerings, standard notations, and naming conventions.

Adherence to standard templates, using a finite set of allowable keywords, will make the automation of documentation assessment a straightforward exercise. We recommend developing documentation templates that follow OMG documentation standards, for example, or any other type of standard that the customer would like its developers to adhere to (ISO, IEEE, and so forth).

Document Test Cases in a Standard Way

Much time is spent on test case documentation. While some sort of test case documentation is always desired, this process can be automated partially, if automated test case generation from use cases or models, for example, is allowed. Much research has gone into various technologies that allow test case generation from models, etc., and a standard way of documenting test cases is the goal. Various efforts are under way to develop standards, such as the MOF to Text standard (the Web site is www.omg.org/cgi-bin/doc?formal/08-01-16.pdf) and IBM/Telelogic's Rhapsody, which provides an automated test case generator (ATG) to produce unit test cases.

ATG is developed using a formal methodology to decompose requirements written in a natural language in order to produce a set of unambiguous rules, object relationships, states, and so on that define the rules/behavior described by the requirements document(s). The rules and relationships are captured using a formal language.

The "formal" language description then becomes the blueprint to generate and identify rule dependencies (actions/reactions), which form "threads" of potential sequences. These dependency threads are the basis for test case development (to develop required data sets, system configuration, and event triggers/and system stimulation). ATG has a test driver component that is used to stimulate system sensor and communication interfaces in a controlled or ad hoc environment and monitor responses.

The concept of ATG can be applied to virtually any rule set. Our work has focused on developing test cases and test capability to assess tactical data link

standards, as these standards use a lot of automated processes that can be readily verified using automated test case documentation and test case generation.

We have developed a to-be-patented technology that allows for test case generation using GUI stimulation. This is another way to automate the development of standard test case documentation. A case study of this is provided in Appendix D.

Adhere to Coding Standards

Software generally needs to be cross-platform-compatible and developed in a standardized fashion, in order to allow for maintainability and portability and to be most efficient. For best AST support, it is important that customer developers adhere to coding standards such as those defined in the OMG C++ Language Mapping Specification, version 1.1, June 2003, or the OMG C Language Mapping Specification, June 1999 and other standards, such as ISO/ANSI C++, and so forth.

Use of OMG's IDL

Another example of using standardized documentation is to use the OMG IDL[13] to help define the interfaces. The OMG IDL is also an ISO International Standard, number 14750. The use of IDL allows for automatic code generation, turning a time-consuming and error-prone manual task into an efficient automated task, saving valuable time. This is another concept as part of our recommended AAA practice.

GUI Testing Recommendations

Numerous user interface standards and guidelines exist for developers to adhere to when developing their AUT's GUI. However, their usability is limited because they are numerous—i.e., there are different guidelines depending on technology—conflicts within and among standards documents, redundancy between documents, and a lack of specific guidance. Many GUI builders are available, such as NetBeans, the Eclipse GUI builder, and so forth, each allowing the developer various opportunities to create a GUI but none really providing specific standards to allow for ease of AST.

A great idea for effective GUI generation that can support AST efforts is provided by IBM's Reflexive User Interface Builder (RIB), which builds Java

GUIs simply and quickly with new technology from alphaWorks.[14] It provides the following features:

- RIB specifies a flexible and easy-to-use XML markup language for describing Java GUIs and provides an engine for creating them. You can define color scheme, font, icon usage, menu/functionality placement, etc.
- You can use RIB to test and evaluate basic GUI layout and functionality, or to create and render GUIs for an application. This concept for AST of a GUI application is ideal.

The RIB concept allows for GUIs to be generated in a standard way, which contributes to effective AST.

Some tools use virtual network computing (VNC) technology for AST. As mentioned, capture/playback tools record the test engineer's keystrokes in some type of scripting language and allow for script playback for baseline verification. Often a requirement exists that the capture/playback tool cannot reside on the AUT, and remote technology is required to access it; in this case VNC technology is applied. In the case of the automated tool using VNC technology, the following recommendations should be considered, because these types of capture/playback tools are sensitive to such changes:[15]

- Control "font smoothing" and other text characteristics should not be changed.
- Don't change the color depth of the AUT.
- Display settings need to stay the same (resolution, etc.).
- If possible, keep the default settings in the OS; use standard visual settings.

Many more practices can be applied to support standard GUI generation and to make GUI testing more effective. For example, when a tester clicks on a GUI control, customized code modules get generated that allow for quick and consistent code generation, eliminating manual code entry/development (this is another concept that is part of our AAA practice).

Additionally, we recommend GUI object naming standards, which are discussed next.

GUI Object Naming Standards

Many automation tools key on the names of objects. Not only does this facilitate development of automation test programs, but it also encourages good software development practices. Microsoft, for example, promotes naming standards at http://support.microsoft.com/kb/110264.

Failure of application developers to name objects and then not name them uniquely is certain to delay test automation programming.

Library Concept of Code Reuse

Software consisting of various components that can be checked out, integrated, and reused, as with the library concept of code reuse described in Chapter 1, lends itself to effective AST. The goal here is to apply AST to various code baselines; and when code components are reused, the corresponding automated tests are applied. This code and automated test reuse will allow for automated testing of each component in a shortened time frame, thus improving AST efficiencies.

4.4 The Forest for the Trees—Not Knowing Which Tool to Pick

When scanning online testing user group postings (such as www.sqaforums.com), many times throughout the week, you may see this query: "Which is the best tool for *xyz*?" where *xyz* equals any testing category/tool, such as automated software testing, performance testing, defect tracking, configuration management, security testing, and other tools used to improve the STL. No matter which testing category or tool this specific query is related to, the answer to the question is often surprising to newcomers but generally will be "It depends." There is no one best tool on the market that will fit every organization's needs.

Finding the best tool for an organization requires a detailed understanding of the problem to be solved and depends on the specific needs and requirements of the task at hand. Once the problem is clear, it is possible to start evaluating tools. You can choose from commercial and open-source solutions—or you can decide to build your own.

Commercial solutions certainly have their advantages, including published feature road maps, institutionalized support, and stability (whether real or

perceived). But buying from a software vendor also has its downsides, such as vendor lock-in, lack of interoperability with other products, lack of control over improvements, and licensing costs and restrictions. Those downsides might be applied to some open-source projects, but the advantages of leveraging the open-source community and its efforts are holding sway with more and more companies. Advantages of open-source include

- No licensing fees, maintenance, or restrictions
- Free and efficient support (though varied)
- Portability across platforms
- Modifiable and adaptable to suit your needs
- Comparatively lightweight
- Not tied to a single vendor

Given the large open-source code base that now exists, there is no need to reinvent the wheel and re-create a code base that already exists and has been tested in the field.

How to Evaluate and Choose a Tool

When tasked with selecting the "best" tool for our needs or for a specific client, we generally approach any type of tool evaluation strategically, as follows:

1. Identify the problem we are trying to solve.
2. Narrow down the tool requirements and criteria.
3. Identify a list of tools that meet the criteria.
4. Assign a weight to each tool criterion based on importance or priority.
5. Evaluate each tool candidate and assign a score.
6. Multiply the weight by each tool candidate score to get the tool's overall score for comparison.

Based on the tool evaluation criteria, we assign a weight based on feature importance to each tool criterion—i.e., the more important a tool criterion is for the client, the higher the weight (from 1 to 5)—then rank the various tools. The rank ranges from 1 to 5 and is based on how closely each tool criterion is met by

the tool. Weight and rank are then multiplied to produce a final "tool score." Features and capabilities of candidate tools are then compared based on the resulting tool score in order to determine the best fit.

There are hundreds of choices for each tool category, and it doesn't make sense to implement these steps for 100 tools. Instead, it's a good idea to narrow down the broad tool list to a select few. This can be done using criteria such as the following:

- **Requirements:** Does the tool meet the high-level requirements? For example, if you are looking for a Web-based solution, but the tool works only as client-server, it would not be considered.
- **Longevity:** Is the tool brand-new or has it been around a while? Darwin's principle of "survival of the fittest" applies specifically to the tool market. Decide on a number of years you think the tool should have been in use.
- **User base:** A large user base generally indicates a good utility. With open-source, a busy development community also means more people providing feedback and improvements.
- **Past experience:** In the past, you or your client has had a good experience with a specific defect reporting tool that meets the high-level requirements, longevity, and user base criteria described.

Using these criteria, you can generally narrow down the field from the scores of tools to a smaller subset.

Additionally, when evaluating tools, the following high-level tool quality attributes need to be considered; they are applicable to almost all tools independent of category.

- Ease of product installation
- Cleanliness of uninstall
- Adequacy and responsiveness of the support team; also, available user groups to help answer questions, etc.
- Completeness and comprehensibility of documentation
- Configurability—the ability to be adapted to each evaluation activity; refers to how easy it is to set up each new evaluation project

- Tuneability, or the ability to guide and focus the analysis toward specific desired features, flaw types, or metrics; refers to how easy it is to fine-tune or tailor different assessments of the same project
- Integratability/interoperability—the level of integration into a larger framework or process supported by the tool, if that's part of the plan
- Balance of effort—the ratio of tool analysis to human analysis in finding actual flaws; refers to how much information the tool needs from a human to complete its analysis
- Expandability—whether the tool suite works on various applications and infrastructures
- Extensibility and technology lag—all commercial tools will eventually experience a technology lag behind the architectures they are targeted to support. When new development architectures are revised to add new features for software developers, there is a good chance that test automation tools may not recognize new objects. Therefore, it is important to evaluate tool extensibility and/or flexibility when dealing with new technology, objects, or methods.

These are some areas to consider when picking a tool. Appendix C provides actual examples for picking the right tool.

4.5 Lack of Automation Standards across Tool Vendors

Numerous vendors provide AST tools, and various open-source testing tools are available, but a lack of automation standards persists.

Many different types of standards have the potential to affect AST. Improved ROI from automated testing can be realized through standards applied to the component(s) under test, the test tools and harness, and other aspects of the test environment. Key considerations in determining the types of standards of greatest interest include the degree to which standards of that type support the following characteristics:

- **Ease of automation**—reduction of the time and complexity needed for automation, resulting in a reduction of the initial investment or an increase in the degree of automation that can be achieved

- **Plug and play**—increased reuse of automated test patterns across products, allowing for reusability of various automation components given the same test scenario
- **Product availability**—increased selection of products supporting automated testing, including test tools as well as other related capabilities such as products to support monitoring and control of the application during testing
- **Product interchangeability**—reduction of vendor lock-in, enabling developers to choose different automation tools for different parts of the testing process or for different baselines, while leveraging prior automation efforts
- **Product interoperability**—ability to use multiple products within a single test set, enabling developers to leverage the capabilities provided by multiple products, resulting in a more robust and higher-quality test
- **Cross-platform compatibility**—ability of one or more tools to be cross-platform-compatible across various OSs and technologies
- **Testing capability**—improved robustness and thoroughness of automated testing, resulting in higher-quality tests

Sample Automated Test Tool Standards

Automated test tools provide many opportunities for standardization. Currently, most automated test standards address hardware testing. However, significant benefits could be gained by standardizing various aspects of software test tools.

- **Scripting language:** Automated test tools use a scripting language to control the events of the tests. Each test tool comes with its own scripting language. Although some of these languages conform to open standards, others are proprietary. A standard for a common scripting language would improve product interchangeability. A set of scripts that work with one automated testing tool could potentially be used with others. For example, currently, even if tool A uses the same scripting language as tool B, the scripts are not interchangeable, because tool A and tool B have different recording mechanisms.[16]
- **Capture feature:** Many automated testing tools come with a capture feature, where the testers' keystrokes are recorded as they execute a test.

All automated testing tools have different and unique recording features. None of the features are standardized, thus producing different script outputs for the same testing steps.

- **Test data:** Many automated testing tools provide a way to generate test data. Some provide a test data database, others provide a flat file, but rarely provide both. A standardized way of generating test data would be useful.
- **Modularity:** Currently the various automated testing tools provide different features to allow for test script componentizing and modularity; i.e., a subscript can be called from a superscript. Providing a modularity feature that's standardized across tools would allow for reusability of modular and componentized scripts.
- **APIs:** Many automated tests require a test harness; i.e., the test harness receives inputs from the test tool via an application program interface (API) that is specific to the tool. It converts the inputs into API calls that are specific to the application being tested, effectively serving as a bridge between the test tool and the AUT. The harness is also responsible for collecting the results and providing them back to the test tool. The API between the test tool and the test harness is not currently standardized. Standardization of this API would be another significant step toward enabling interchangeability of automated test tools. The consistency provided by common scripting languages and APIs would also provide greater ease of use and would reduce the learning curve associated with each new test tool.
- **Test tool output and reporting:** Many automated testing tools currently use various methods to produce test results output and reporting. No standard way of test-run output and reporting exists.
- **Integration with other testing support tools:** Many automated testing tools provide APIs that allow imports and/or exports to other test support tools, such as requirements management, configuration management, and defect tracking tools. However, not all tools provide this standard capability.

In addition to the standards described here, other areas can benefit from standardization to support the automation cause, such as systems engineering standards to be applied during software development in support of AST. Cur-

rently an effort is under way at OMG, which IDT helped initiate, to standardize some of the AST efforts, called automated test and retest (ATRT).

4.6 Lack of Business Case

In Chapter 3 we discussed the need for a business case. Our experience, described there, revealed that if the business case has been developed, is approved, and buy-in exists from all stakeholders regarding the automated testing effort, then everyone will feel responsible for its success. If all contribute to the success of the automated testing program, chances for success are much higher. It is therefore an important goal to develop the business case and get approval. See Chapter 3 for how to develop a business case.

Summary

Even though most agree that AST is an efficient tool or practice in the testing toolbox, AST efforts still fail for various reasons. R&D needs to consider not only research and development but associated testing concepts, since testing can have a big impact on the success of technology implementation and/or failure of a company's perceived quality.

Many AST myths still persist, and the realities of AST have been described here. It is necessary to manage expectations accordingly.

If software developers follow selected best practices, such as the Open Architecture standards described here, or use our proposed concept of AAA, use standard documentation, and abide by some of the GUI/interface recommendations listed here, as well as follow selected best practices such as the library concept of component reuse and coding standards adherence, AST efforts stand to succeed and can be seamlessly integrated into an already effective process. Vendors could work on automated tool standards across tools, which additionally could contribute to the success of AST efforts.

Notes

1. Dustin et al., *Automated Software Testing*.
2. http://video.google.com/videoplay?docid=8774618466715423597&hl=en.

3. www.businessweek.com/magazine/content/08_17/b4081061866744.htm?chan=magazine+channel_special+report.
4. Described at www.sdtimes.com/content/article.aspx?ArticleID=32098.
5. www.businessweek.com/magazine/content/08_17/b4081061866744.htm?chan=magazine+channel_special+report.
6. Dustin et al., *Automated Software Testing.*
7. Kenneth H. Rosen, *Discrete Mathematics and Its Applications* (McGraw-Hill, 1991).
8. Ibid.
9. Contributed by J. Offutt. See also P. Ammann and J. Offutt, *Introduction to Software Testing* (Cambridge University Press, 2008).
10. www.omg.com/.
11. www.nswc.navy.mil/wwwDL/B/OACE/.
12. Dustin et al., *Automated Software Testing.*
13. www.omg.org/gettingstarted/omg_idl.htm.
14. www.ibm.com/developerworks/java/library/j-rib/.
15. Recommendations from the makers of Eggplant; see www.testplant.com.
16. We verified this by comparing two major vendor-provided testing tools using the same scripting languages. Too much proprietary information is included in each script to make it transferable.

Index

Page numbers followed by a t or and f indicate a table or a figure, respectively.

B

S

T

U

V

W